O'REILLY®


图灵程序设计丛书



Java 数据科学实战

Data Science with Java

提升数据科学实战技能，打造属于你的Java数据科学项目

[美] 迈克尔·R.布茹斯托维奇 著
姜建锦 赵绪营 张岩 译


中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

姜建锦

清华大学博士，北京电子科技学院网络空间安全系教师，目前主要从事计算机系统结构、分布式系统等的研究及相关教学工作。

赵绪营

中科院博士，北京电子科技学院网络空间安全系教师，目前主要从事生物特征识别与加密，计算机视觉的研究及相关教学工作。

张岩

博士，北京电子科技学院网络空间安全系副教授，目前主要从事信息物理融合系统、软件系统建模和验证的研究及相关教学工作。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。


图灵程序设计丛书

Java数据科学实战

Data Science with Java

[美] 迈克尔 • R. 布茹斯托维奇 著
姜建锦　赵绪营　张岩 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　　京

图书在版编目（CIP）数据

Java数据科学实战 / (美) 迈克尔·R.布茹斯托维奇著 ; 姜建锦, 赵绪营, 张岩译. -- 北京 : 人民邮电出版社, 2020.2
（图灵程序设计丛书）
ISBN 978-7-115-53330-2

Ⅰ. ①J… Ⅱ. ①迈… ②姜… ③赵… ④张… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2020)第010913号

内 容 提 要

本书基于清晰的、面向对象的 Java 代码，讨论了数据科学研究的一些基本原理。考虑到项目所需的可伸缩性、稳健性以及便利性，Java 是一门理想的语言。本书解释了数据科学过程每个步骤背后的基本数学原理，以及如何将这些概念应用于 Java。本书内容涉及数据输入与输出、线性代数、统计学、数据操作、学习与预测，以及 Hadoop MapReduce 在这个过程中所扮演的关键角色。书中还提供了在应用程序中使用的代码示例。

本书适合数据科学工作者以及希望提高数据科学技能的 Java 软件工程师阅读。

◆ 著　　[美] 迈克尔·R.布茹斯托维奇
译　　姜建锦　赵绪营　张　岩
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：12.5
字数：295千字　　　　2020年2月第1版
印数：1－3 500册　　　2020年2月北京第1次印刷

著作权合同登记号　图字：01-2019-7538号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

© 2017 by Michael Brzustowicz.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2020. Authorized translation of the English edition, 2020 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2017。

简体中文版由人民邮电出版社出版，2020。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

谨以此书献给我的合伙人，以及我们的两家创业公司。

目录

前言

数据科学是一个多样化且正在发展的领域，它涉及数学与计算机科学的许多子领域。在数据科学家所研究的领域中，多种学科交织在一起，统计学、线性代数、数据库、机器智能以及数据可视化仅是其中的一部分。各种技术大量存在，用于数据科学实践的工具也正在快速演化。本书基于清晰的、面向对象的 Java 代码，主要讨论一些核心的基本原理。本书将激励你立刻着手实践数据科学技能，希望你可以在开发下一代数据科学技术时处于领先位置。

读者对象

本书面向的是那些已经熟悉应用开发概念的科学家和工程师，他们想直接从事数据科学研究。本书将循序渐进地引导读者进入数据科学的工作流程，在解释数学原理的同时给出代码示例。对于想深入学习数据科学的读者而言，本书是个完美的出发点。

写作初衷

我撰写本书是为了开启一项运动。由于 R 语言和 Python 语言的推动，数据科学迅速成为热门研究领域，但很少有数据科学从业人士冒险涉足 Java 世界。显然，数据探索工具适合采用解释型语言，但是在工程与科学混合的领域，必须综合考虑可伸缩性、稳健性以及便利性。Java 也许正是那种能够满足上述所有要求的语言。如果本书对你有所鼓舞，那么期待你可以向众多支持数据科学的 Java 开源项目贡献代码。

数据科学现状

数据科学正在不断变化，包括其应用范围以及实践数据科学的人。技术的发展非常快，仅需要几年甚至几个月的时间，顶级的算法就会过时。对于实际的解决方案，人们抛弃了长期采用的标准化做法。成功道路上的障碍通常是由定量科学未曾涉及的领域内的人士克服

的。目前，数据科学已经是一门本科生课程了。在未来，要想取得成功，只有一条途径，即掌握数学、熟悉代码，并知悉所要解决的问题。

本书导读

本书是一场穿越数据科学工作流程的逻辑之旅。第 1 章讨论获取数据、清理数据以及以最纯粹的方式排列数据的众多方法，并讨论把数据输出至文件或者进行绘制的基本方法。第 2 章讨论把数据视为矩阵的重要概念，这一章将详细回顾矩阵的运算。有了数据并且知道数据应当采用何种数据结构后，第 3 章引入测试数据来源和有效性的基本概念。第 4 章直接使用第 2 章和第 3 章的概念，把数据转换为稳定、可用的数值。第 5 章介绍一些实用的监督型学习算法与无监督型学习算法，以及评估这些算法是否成功的方法。第 6 章提供快速指南，采用适合数据科学算法的定制组件，设置并运行 MapReduce 任务。附录 A 给出了一些有用的数据集。

排版约定

本书使用了下列排版约定。

黑体

表示新术语或重点强调的内容。

等宽字体（`constant width`）

表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体（**`constant width bold`**）

表示应该由用户输入的命令或其他文本。

等宽斜体（*`constant width italic`*）

表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/oreillymedia/Data_Science_with_Java 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN，比如“*Data Science with Java* by Michael Brzustowicz (O'Reilly). Copyright 2017 Michael Brzustowicz, 978-1-491-93411-1”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（前身为 Safari Books Online）是一个会员制的培训和参考平台，面向企业、政府、教育从业者和个人。

Safari 用户可访问 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等 250 多家出版社的上千种图书、培训视频、学习路径、交互式教程和精选播放列表。

如需了解更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。[1]

注 1：可以访问本书图灵社区页面（https://www.ituring.com.cn/book/2082）下载示例代码并提交中文版勘误。
——编者注

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

我要感谢本书的编辑，O’Reilly 出版公司的 Nan Barber 与 Brian Foster，感谢他们在写作过程中不断给予我激励与指导。

我还要感谢 O’Reilly 出版公司的工作人员 Melanie Yarbrough、Kristen Brown、Sharon Wilkey、Jennie Kimmel、Allison Gillespie、Laurel Ruma、Seana McInerney、Rita Scordamalgia、Chris Olson 以及 Michelle Gilliland，他们都为本书的出版做出了贡献。

本书得益于我的同事 Dustin Garvey、Jamil Abou-Saleh、David Uminsky 以及 Terence Parr 的许多技术评论与主张。我真诚地感激你们的帮助。

电子书

扫描如下二维码，即可购买本书中文版电子版。



第 1 章

数据的输入与输出

我们身边每时每刻都有事件发生。有时候，我们记录下了时空中某个特定点发生的离散事件。然后，我们可以把**数据**（data）定义为一系列的记录，这些记录是某人（或某物）花时间以任何可以想象出的格式写下来（或者呈现出来）的。数据科学家面对的数据有文件、数据库、Web 服务，等等。通常，人们在定义能够准确地表示各种变量的名称、类型、变量值的范围以及这些变量之间关系的模式或者数据模型时，会遇到许多困难。然而，在获取数据时，并非总能强行采用某种模式。真实的数据（即使是精心设计的数据库）通常会存在以下问题：缺失值、拼写错误、不正确的格式化类型、对同一值的重复表示，甚至将几个变量拼接成了一个变量。尽管人们对实现机器学习算法以及创建漂亮图形感到兴奋，但是，数据科学中最重要且最耗时的工作是准备数据和确保其完整性。

1.1 究竟何谓数据

你的最终目标是从源头获取数据，通过统计分析或者学习对数据进行归约，然后根据学到的东西给出某种知识——通常采用图形的形式表示。然而，即使所求结果是单个值，例如总收益、参与度最高的用户或者品质因数，也需要遵循相同的流程：**输入数据**（input data）→**归约分析**（reductive analysis）→**输出数据**（output data）。

鉴于实际的数据科学受业务问题驱动，因此从右向左来看这个流程会更方便一些。首先，把试图回答的问题正式确定下来。例如，需要的是按地区排列的顶级用户列表、关于下周每天收入的预测，还是库存物品之间相似性分布的图示？下一步，进行一系列分析，以回答这些问题。最后，既然已经选定解决问题的方法，那么为了实现这个目标，究竟需要哪些数据？你会惊奇地发现自己没有所需要的数据。另外，通常你会发现，一些比预想的简

单得多的分析工具就足以得到所需的输出。

本章将探究从各种数据源读取数据以及写入数据的具体细节。重要的是，需要问自己：对于后续每个步骤，需要什么样的数据模型？也许，为了容纳这些数据，建立一系列的数值数组类型（例如 `double[][]`、`int[]`、`String[]`）就足够了。这样做可能是有益处的：创建容器类以保存每条数据记录，然后把这些对象添加到 `List` 或 `Map` 中。还有另外一种有用的数据模型，即在 JavaScript 对象表示法（JSON）文档中，把每条记录都表示为键 - 值对的集合。具体应该采用哪种数据模型，很大程度上取决于后续数据消费过程的输入需求。

1.2　数据模型

数据采用哪种形式，需要转换成什么形式，才能继续往下进行？假设文件 somefile.txt 包含数行数据，每一行都有 `id`、`year` 以及 `city` 数据。

1.2.1　一维数组

对于这个示例而言，最简单的数据模型是为 `id`、`year` 以及 `city` 这 3 个变量创建一系列的数组：

```
int[] id = new int[1024];
int[] year = new int[1024];
String[] city = new String[1024];
```

由于 `BufferedReader` 循环遍历该文件的每一行，因此借助于增量计数器，可以把变量的值放置到各个数组的相应位置中。对于已知维度的洁净数据，这种数据模型可能就足够了，此时所有代码都放入同一个可执行类中。可以直接把这个数据提供给任何数量的统计分析算法或者学习算法。然而，有时可能需要将代码模块化，并构建适合于数据源与数据模型的每种组合的类以及随后的方法。在这种情况下，为了适应新的参数而必须改变现有方法的签名时，在数组之间切换会变得困难。

1.2.2　多维数组

若想每一行容纳一条记录的所有数据，则这些数据必须是相同的类型。所以，在上面的例子中，只有把城市赋值为整型值，才能工作。

```
int[] row1 = {1, 2014, 1};
int[] row2 = {2, 2015, 1};
int[] row3 = {3, 2014, 2};
```

也可以将其处理为二维数组。

```
int[][] data = {{1, 2014, 1}, {2, 2015, 1}, {3, 2014, 2}};
```

第一次查看数据集时，可能已经有复杂的数据模型，或者只是文本、整数、double 型以及日期时间型数据的混合体。理想情况下，当确定了要将哪些数据输入到统计分析算法或者学习算法中时，这些数据已经被转换为二维数组，数组的元素是 double 型。然而，需要大量工作才能达到这一步。一方面，以矩阵的形式提供数据，从而采用机器学习方法取得进展是方便的；另一方面，可能不知道需要做哪些折中，或者已经扩散了哪些未被检测到的错误。

1.2.3 数据对象

另一种选择是创建容器类，然后把这些容器对象添加到 List 或 Map 等集合中。这样做的优点是，将一条特定记录的所有值保持在一起，向类中添加新成员时，不会破坏任何以这个类作为参数的方法。文件 somefile.txt 中的数据可以用下面的类表示：

```
class Record {
    int id;
    int year;
    String city;
}
```

确保该类尽可能是轻量级的，因为包含这些对象的集合（List 或 Map）会形成大的数据集。理想情况下，任何作用在 Record 对象上的方法应该都是其所属类（该类可能命名为 RecordUtils）中的静态方法。

集合结构 List 用来存放所有 Record 对象。

```
List<Record> listOfRecords = new ArrayList<>();
```

采用 BufferedReader 遍历数据文件时，会对每一行进行解析，并将其内容存放在新的 Record 实例中，再把每个新的 Record 实例添加到 List<Record> listOfRecords 中。若需要通过键进行快速查找，并获取某个单独的 Record 实例，则可以使用 Map。

```
Map<String, Record> mapOfRecords = new HashMap<>();
```

对于特定记录而言，其键应当是唯一的标识符，例如记录编号或者 URL。

1.2.4 矩阵和向量

矩阵（matrix）和**向量**（vector）是更高层次的数据结构，它们分别由二维数组与一维数组构成。通常，数据集包含多个行与列，可以说这些变量形成了二维数组（或**矩阵**）$\boldsymbol{X}$，其中有 m 行 n 列。若选择 i 作为行索引、j 作为列索引，则 $m \times n$ 矩阵的每个元素是 $x_{i,j}$。

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix}$$

把值放入类似于矩阵的数据结构中，可以获得便利。在多种情况下，要对数据进行数学运算。矩阵实例可以拥有完成这些运算的抽象方法，也有适合于手头某项任务的实现细节。第 2 章会具体讨论矩阵和向量。

1.2.5 JSON

JavaScript 对象表示法（JavaScript Object Notation，JSON）已经成为表示数据的一种流行格式。通常，JSON 数据采用 json.org 列举的简单规则来表示：采用双引号。结尾处没有逗号。JSON 对象的外层采用了花括号，内部是任意有效的键 - 值对集合，这些键 - 值对用逗号分隔（由于不能保证其内容的顺序，因此将其当作 `HashMap` 类型来处理）。

```
{"city":"San Francisco", "year": 2020, "id": 2, "event_codes":[20, 22, 34, 19]}
```

JSON 数组的外层是方括号，其内部的有效 JSON 内容采用逗号分隔（数组内容的顺序是有保证的，因此当作 `ArrayList` 类型来处理）。

```
[40, 50, 70, "text", {"city":"San Francisco"}]
```

有两类 JSON 数据结构。一些数据文件包含完整的 JSON 对象或数组，这些通常是配置文件。然而，另一类常见数据结构是由独立 JSON 对象构成的文本文件，每个 JSON 对象占据一行。注意，严格来说，这种类型的数据结构（由 JSON 对象构成的列表）并不是 JSON 对象或者数组，原因是文件中并没有闭合的大括号，或者是由于行之间没有逗号。正因如此，试图把整个数据结构解析为 JSON 对象或数组会失败。

1.3 处理实际数据

实际数据往往是杂乱、不完整、不正确的，有时甚至是不连贯的。若正在处理的是“完美”的数据集，那是因为有人已经花费大量时间与精力使它变成那个样子。也可能是，数据事实上并不完美，而你正在不知不觉地对垃圾数据进行分析。唯一可靠的途径是从源头采集数据，然后自己处理。这样，若发生了错误，也可以知道应该由谁负责。

1.3.1 空值

空值以多种形式出现。若数据在 Java 内部传递，则完全可能出现空值。如果正从文本文件解析字符串，那么空值可能会表示为多种形式的字符串，包括 `"null"`、`"NULL"` 或 `"na"` 等其他字符串，甚至是句点。在任何一种情况下（空值或者 null 字符串），都需要对其进行记录。

```
private boolean checkNull(String value) {
    return value == null || "null".equalsIgnoreCase(value);
}
```

通常，空值已经被记录为一个空格或者一系列空格。尽管这样做有时让人烦恼，但是它可以达到某种目的，因为把数据点不存在这一概念编码为 0 并不总是合适的。例如，如果在记录二值变量（0与 1）时遇到了一个并不知道其值的项，那么为其错误地赋值为 0（并写到了文件中），就会错误地把真的负例赋给这个项。在向文本文件写入空值时，我的偏好是写入长度为 0 的字符串。

1.3.2　空格

空格广泛存在于实际数据中。采用 `String.isEmpty()` 方法可以直接检查字符串是否为空字符串。然而请注意，由空格组成的字符串（即使只有单个空格）并不是空的！首先，用 `String.trim()` 方法移除那些位于输入值开头部分或结尾部分的空格，然后检查它的长度。只有在字符串长度为 0 时，`String.isEmpty()` 才返回 `true`。

```
private boolean checkBlank(String value) {
    return value.trim().isEmpty();
}
```

1.3.3　解析错误

一旦知道了字符串的值既不是空值也不是空格，就将其解析为所需要的类型。这里不讨论如何将字符串解析为字符串，因为并没有什么需要解析的。

当处理数值类型时，把字符串直接转换为基本类型，例如 `double`、`int` 或 `long`，是不明智的。推荐做法是采用对象封装类，例如 `Double`、`Integer` 和 `Long`，它们都有字符串解析方法。如果发生错误，那么这些方法会抛出 `NumberFormatException` 异常。可以捕获该异常，并更新解析错误计数器；也可以打印该错误，或将其记入日志中。

```
try {
    double d = Double.parseDouble(value);
    // 处理d
} catch (NumberFormatException e) {
    // 对解析错误计数器进行递增等操作
}
```

同样，也可以用 `OffsetDateTime.parse()` 方法解析日期时间格式的字符串。如果在输入字符串中发生错误，那么可以捕获到 `DateTimeParseException` 异常，并记入日志中。

```
try {
    OffsetDateTime odt = OffsetDateTime.parse(value);
    // 处理odt
} catch (DateTimeParseException e) {
    // 对解析错误计数器进行递增等操作
}
```

1.3.4 异常值

清理并解析数据之后，即可检查其值是否符合需求。如果期望值是 0 或 1，得到的却是 2，那么显然这个值超出了范围，可以把这个数据点标记为异常值。就如同在处理空值与空格时所做的那样，可以对这个值进行布尔测试，以决定它是否处于可接受的值范围之内。这种做法对于数值类型、字符串以及日期时间类型都是适合的。

对数值类型进行范围检查时，需要知道可接受的最大值与最小值，以及这两个值是否包括在内。例如，若设定 `minValue = 1.0` 且 `minValueInclusive = true`，则所有大于或等于 1.0 的值都将通过测试。如果设定 `minValueInclusive = false`，则只有大于 1.0 的那些值才会通过测试。代码如下：

```
public boolean checkRange(double value) {
    boolean minBit = (minValueInclusive) ? value >= minValue : value > minValue;
    boolean maxBit = (maxValueInclusive) ? value <= maxValue : value < maxValue;
    return minBit && maxBit;
}
```

可以为整型数据写出类似的方法。

也可以通过设置有效字符串的枚举，来检查字符串的值是否处于可接受范围之内。为此，可以通过创建有效字符串的 `Set` 实例（例如 `validItems`）来实现，其中的 `Set.contains()` 方法用于测试输入值的有效性。

```
private boolean checkRange(String value) {
    return validItems.contains(value);
}
```

对于 `DateTime` 对象，可以检查日期是否在最小日期之后，在最大日期之前。在这种情况下，把日期的最小值与最大值定义为 `OffsetDateTime` 对象，然后测试输入的日期时间是否介于最小值与最大值之间。注意，`OffsetDateTime.isBefore()` 与 `OffsetDateTime.isAfter()` 是不包括边界值的。如果输入的日期时间等于最小值或最大值，那么测试将返回 `false`。代码如下：

```
private boolean checkRange(OffsetDateTime odt) {
    return odt.isAfter(minDate) && odt.isBefore(maxDate);
}
```

1.4 管理数据文件

数据科学艺术始于对数据文件的管理。选择如何构建数据集不仅关系到效率，而且还关系到灵活性。读写文件有多种选择，最基本的方法是采用 `FileReader` 实例把整个文件的内容读入到 `String` 类型中，然后把这个 `String` 解析为相应的数据模型。对于大文件而言，采用 `BufferedReader` 分别读文件的每一行，可以避免输入输出错误。这里的策略是在读每一

行时进行解析，仅保留那些需要的值，并把这些记录填充到数据结构中。如果每行有 1000 个变量，而只需要其中的 3 个，则没必要保存所有变量。同样，若某一行中的数据不符合某个标准，则也没有必要保存该行。对于大型数据集，相比于把所有行读入到字符串数组（String[]）中，随后再进行解析，这种方法可以节省很多资源。在管理数据文件的这一步骤中，考虑得越多，收获就越大。不论是统计、学习还是绘制，这些后续的每个步骤都将依赖于构建数据集时的决策。俗话说“输入的是无用数据，那么输出的也一定是无用数据”，这绝对在理。

1.4.1　首先理解文件内容

数据文件的结构繁杂，可能会呈现一些不符合需要的**特征**（feature）。回忆一下，ASCII 文件仅是打印到每一行上的一些 ASCII 字符的集合。它无法保证数值的格式或精度，也不能保证是采用单引号还是双引号，以及是否包含大量空格、空值以及换行符。简而言之，不管对文件内容做了怎样的假设，文件每一行所包含的内容都可能是各式各样的。用 Java 读取文件之前，先用文本编辑器或命令行看一下该文件。注意每一行中每一项的个数、位置以及类型，要特别关注缺失的值或空值是如何表示的。同样，要注意分隔符的类型，以及任何描述数据的头部。若文件足够小，则可以通过肉眼来查看哪些行有缺失或格式错误。例如，假定在 bash shell 中采用 UNIX 命令 less 来查看 somefile.txt 文件：

```
bash$ less somefile.txt

"id","year","city"
1,2015,"San Francisco"
2,2014,"New York"
3,2012,"Los Angeles"
...
```

那么可以看到数据集的格式采用的是用逗号分隔的值（CSV），它由 id、year 以及 city 这 3 列组成，可以快速地检查文件的总行数。

```
bash$ wc -l somefile.txt
1025
```

这表明文件共有 1024 行数据，再加上一行头部。其他格式也是可能的，例如以制表符分隔的值（TSV）、所有值都拼接在一起的“大字符串”格式，以及 JSON。对于大文件，也许想看前 100 行左右，并把它们转换为一个摘要文件，以便于应用的开发。

```
bash$ head -100 filename > new_filename
```

在某些情况下，文件太大了，以至于无法用一双眼睛来细看以发现其结构或错误。显然，在检查具有 1000 列的数据文件时，会遇到障碍。同样，也不太可能通过滚动有 100 万行数据的文件来发现其中的格式错误。在这种情况下，必须有现存的数据字典，用以描述列的格式，以及每一列预期的数据类型（例如整型、浮点型、文本）。采用 Java 对文件进行

解析时，可以用编程方式检查每一行数据。这期间，程序可能会抛出异常，而且也许会将不合规则的那些行的内容全部打印出来，以便于检查是什么出错了。

1.4.2 读取文本文件

读取文本文件的通常做法是创建 FileReader 实例，其外层是 BufferedReader，这样可以读取每一行。在这里，FileReader 的参数是 String 类型的文件名，但是 FileReader 也可以采用 File 对象作为它的参数。当文件名和路径依赖于操作系统时，File 对象是有用的。下面是采用 BufferedReader 从文件中按行读取的一般形式：

```
try(BufferedReader br = new BufferedReader(new FileReader("somefile.txt")) ) {
    String columnNames = br.readline(); // 若文件存在，则仅执行这一操作
    String line;
    while ((line = br.readLine()) != null) {
        /* 解析每一行 */
        // TODO
    }
} catch (Exception e) {
    System.err.println(e.getMessage()); // 或记录错误
}
```

若文件在远程的某地，则也可以做同样的事情。

```
URL url = new URL("http://storage.example.com/public-data/somefile.txt");
try(BufferedReader br = new BufferedReader(
    new InputStreamReader(url.openStream())) ) {
    String columnNames = br.readline(); // 若文件存在，则仅执行这一操作
    String line;
    while ((line = br.readLine()) != null) {
        // TODO，解析每一行
    }
} catch (Exception e) {
    System.err.println(e.getMessage()); // 或记录错误
}
```

此处只需要关注如何解析每一行。

1. 解析大字符串

考虑某个文件，其中每一行是一个“大字符串”，它由一些值以及任意子字符串拼接而成，这些具有起始位置与结束位置的子字符串对特定变量进行编码。

```
0001201503
0002201401
0003201202
```

前四位数字是编号（id）值，接下来的四位是年份（year），最后两位是城市（city）编码。注意，每一行都可能有几千个字符那么长，字符子串的位置至关重要。典型的情况是，编号用 0 填充，空值用空白表示。注意浮点数中的句点（例如 32.456）被记作空格，

和其他“奇怪的”字符一样！通常，文本字符串采用数值编码。例如，在上面例子中，`NewYork = 01`、`Los Angeles = 02` 以及 `San Francisco = 03`。

此时，可以用 `String.substring(int beginIndex, int endIndex)` 方法访问每一行的值。注意，子字符串从 `beginIndex` 位置开始，到（但不包括）`endIndex` 位置结束。

```
/* 解析每一行 */
int id = Integer.parseInt(line.substring(0, 4));
int year = Integer.parseInt(line.substring(4, 8));
int city = Integer.parseInt(line.substring(8, 10));
```

2. 解析用分界符界定的字符串

考虑到电子表格与数据库转储的广泛应用，你很有可能会在某一时刻遇到 CSV 数据集。解析这种文件可能并不那么容易。假设示例中的数据采用 CSV 文件格式：

```
1,2015,"San Francisco"
2,2014,"New York"
3,2012,"Los Angeles"
```

然后要做的是，用 `String.split(",")` 方法解析，并用 `String.trim()` 方法去掉任何位于开始与结尾处的令人烦恼的空格。同时，有必要采用 `String.replace("\"", "")` 方法去除字符串两边的引号。

```
/* 解析每一行 */
String[] s = line.split(",");
int id = Integer.parseInt(s[0].trim());
int year = Integer.parseInt(s[1].trim());
String city = s[2].trim().replace("\"", "");
```

下面的例子中，文件 somefile.txt 中的数据已经用制表符分隔。

```
1        2015        "San Francisco"
2        2014        "New York"
3        2012        "Los Angeles"
```

通过把前面代码中的 `String[] s = line.split(",")` 替换为 `String[] s = line.split("\t")`，可以对用制表符界定的数据进行分离。

在某一时刻，你一定会遇到有些字段中包含逗号的 CSV 文件。这样的例子包括从用户博客中获取的文本。另一个例子是在一列中出现了不规范的数据，例如“San Francisco, CA”，它没有把城市与州分别放入两个列中。这在解析时非常棘手，并需要正则表达式。但是，为什么不用 Apache Commons CSV 解析库呢？

```
/* 解析每一行 */
CSVParser parser = CSVParser.parse(line, CSVFormat.RFC4180);
for(CSVRecord cr : parser) {
    int id = cr.get(1); // 列从1开始，而不是从0开始
```

```
        int year = cr.get(2);
        String city = cr.get(3);
    }
```

Apache Commons CSV 库也可以处理包括 CSVFormat.EXCEL、CSVFormat.MYSQL 以及 CSVFormat.TDF 在内的常见格式。

3. 解析 JSON 字符串

JSON 是一种用于对 JavaScript 对象进行序列化的协议，可以扩展到所有类型的数据。这种紧凑且易读的格式普遍存在于因特网数据 API 中（特别是 RESTful 服务），并且是许多 NoSQL 解决方案（例如 MongoDB 与 CouchDB）的标准格式。自 9.3 版本开始，PostgreSQL 数据库提供了 JSON 数据类型，可以查询原生 JSON 字段。其显著优势是便于人类阅读，数据结构清晰可见，并且可以“优质打印”。就 Java 而言，JSON 不过是 `HashMaps` 与 `ArrayLists` 的集合而已，可以采用能够想象到的任何嵌套配置。对于前面示例的每一行数据，可以把那些值放入键 - 值对中，从而成为 JSON 字符串格式；字符串采用双引号（而不是单引号）括起来，并且不允许尾端出现逗号。

```
{"id":1, "year":2015, "city":"San Francisco"}
{"id":2, "year":2014, "city":"New York"}
{"id":3, "year":2012, "city":"Los Angeles"}
```

注意，从技术上讲，整个文件本身不是 JSON 对象，把整个文件本身解析为 JSON 对象会失败。要成为有效的 JSON 格式，每一行必须用逗号分隔，然后整个组采用方括号括起来，这将形成 JSON 数组。然而，写出这样的结构是低效且无用的。更方便且更有用的做法是用字符串表示的 JSON 对象构成按行组织的栈。注意，JSON 解析器并不知道键 - 值对中值的类型。因此，要获得它的 `String` 表示，然后采用装箱方法将其解析为基本类型。现在，采用 `org.simple.json` 可以直接构造数据集。

```
/* 在while循环外创建JSON解析器 */
JSONParser parser = new JSONParser();
...

/* 对解析得到的字符串进行类型转换，以创建对象 */
JSONObject obj = (JSONObject) parser.parse(line);
int id = Integer.parseInt(j.get("id").toString());
int year = Integer.parseInt(j.get("year").toString());
String city = j.get("city").toString();
```

1.4.3 读取JSON文件

本节讨论包含字符串化的 JSON 对象或数组的文件。需要提前知道文件是 JSON 对象还是数组。例如，如果在命令行中采用 `ls` 命令查看文件，将看到该文件中是包含花括号（对象）还是方括号（数组）。

```
{{"id":1, "year":2015, "city":"San Francisco"},
 {"id":2, "year":2014, "city":"New York"},
 {"id":3, "year":2012, "city":"Los Angeles"}}
```

然后采用 Simple JSON 库。

```
JSONParser parser = new JSONParser();
try{
    JSONObject jObj = (JSONObject) parser.parse(new FileReader("data.json"));
    // TODO，对jObj对象进行处理
} catch (IOException|ParseException e) {
    System.err.println(e.getMessage());
}
```

若它是数组：

```
[{"id":1, "year":2015, "city":"San Francisco"},
 {"id":2, "year":2014, "city":"New York"},
 {"id":3, "year":2012, "city":"Los Angeles"}]
```

则可以解析整个 JSON 数组。

```
JSONParser parser = new JSONParser();
try{
    JSONArray jArr = (JSONArray) parser.parse(new FileReader("data.json"));
    // TODO，对jObj对象进行处理
} catch (IOException|ParseException e) {
    System.err.println(e.getMessage());
}
```



如果某个文件中确实每一行都是 JSON 对象，那么严格来说，该文件并不是合格的 JSON 数据结构。请参考 1.4.2 节，在那里读取文本文件时，是按照一次解析一行的方式来解析 JSON 对象的。

1.4.4 读取图像文件

使用图像作为学习算法的输入时，需要把图像格式（例如 PNG）转换为合适的数据结构，例如矩阵或向量。这里有几点需要注意。首先，图像是一个二维数组，它包括坐标 $\{x_1, x_2\}$，以及对应的颜色或强度值集合 $\{y_1\cdots\}$，该集合的元素可以用单个整型值存储。若所需的只是以二维整型数组存储的原始数值（这里标记为 `data`），则可以用下面的代码读入这个缓冲的图像：

```
BufferedImage img = null;
try {
    img = ImageIO.read(new File("Image.png"));
    int height = img.getHeight();
    int width = img.getWidth();
    int[][] data = new int[height][width];
```

```
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                int rgb = img.getRGB(i, j); // 负整数
                data[i][j] = rgb;
            }
        }
    } catch (IOException e) {
        // 处理异常
    }
```

通过对整数进行移位操作，可以将其转换为 RGB 分量（红、绿、蓝）：

```
int blue = 0x0000ff & rgb;
int green = 0x0000ff & (rgb >> 8);
int red = 0x0000ff & (rgb >> 16);
int alpha = 0x0000ff & (rgb >> 24);
```

然而，可以用下面的方法从光栅中方便地获取该信息：

```
byte[] pixels = ((DataBufferByte) img.getRaster().getDataBuffer()).getData();
for (int i = 0; i < pixels.length / 3 ; i++) {
    int blue = Byte.toUnsignedInt(pixels[3*i]);
    int green = Byte.toUnsignedInt(pixels[3*i+1]);
    int red = Byte.toUnsignedInt(pixels[3*i+2]);
}
```

颜色可能并不重要，灰度级或许正是所需要的：

```
// 把RGB转换为灰度级(0~1)，其中颜色值的范围是0~255
double gray = (0.2126 * red + 0.7152 * green + 0.0722 * blue) / 255.0
```

此外，在某些场合，二维表示并不是必需的。通过把矩阵的每一行拼接到新向量上，可以把矩阵转换为向量，例如，$\boldsymbol{x}_n = \boldsymbol{x}_1, \boldsymbol{x}_2, \cdots$，其中向量维数 n 等于矩阵的行数乘以列数，即 $m \times p$。在著名的手写图像数据集 MNIST 中，数据已经进行了纠正（居中与裁剪），并转换成二进制格式。因此需要一种专门格式读取该数据（见附录 A），但是它已经是向量（一维）而不是矩阵（二维）格式了。针对 MNIST 数据集的学习技术通常涉及这种向量化的格式。

1.4.5　写入文本文件

把数据写入到文件中的一般方式是采用 `FileWriter` 类，但是依旧要推荐采用 `BufferedWriter` 类，以防止任何输入 / 输出错误。通常做法是把想要写入文件的数据转换为单一的字符串。对于示例中的 3 个变量，可以采用所选择的分界符（要么是逗号，要么是 `\t`）来手动操作。

```
/* 对于每个 Record 记录实例 */
String output = Integer.toString(record.id) + "," +
Integer.toString(record.year) + "," + record.city;
```

在 Java 8 中，`String.join(delimiter, elements)` 方法是方便的。

```
/* 在Java 8中 */
String newString = String.join(",", {"a", "b", "c"});

/* 或者采用Iterator进行迭代 */
String newString = String.join(",", myList);
```

此外，还可以使用 Apache Commons Lang 中的 `StringUtils.join(elements, delimiter)` 方法，或者在循环中使用原生类 `StringBuilder`。

```
/* 在 Java 7 中 */
String[] strings = {"a", "b", "c"};

/* 创建StringBuilder对象并添加第一个成员 */
StringBuilder sb;
sb.append(strings[0]);

/* 略过第一个字符串，因为已经拥有它了 */
for(int i = 1; i < strings.length, i++){
    /* 此处选择分隔符……对于制表符，也可以是\t */
    sb.append(",");
    sb.append(strings[i]);
}

String newString = sb.toString();
```

注意，连续地进行字符串串联操作（`myString += myString_part`）会调用 `StringBuilder` 类，因此也可以直接使用 `StringBuilder` 类（或者不用）。在任何情况下，字符串都是逐行写入的。注意，`BufferedWriter.write(String)` 方法不会写新行。如果想把数据的每条记录都写到单独的一行，则必须调用 `BufferedWriter.newLine()` 方法。

```
try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    for(String s : myStringList){
        bw.write(s);
        /* 不要忘记追加新行！ */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

前面的代码覆盖了文件名指定的文件中所有现存的数据。在某些场合，需要向现存文件追加数据。`FileWriter` 类有可选的布尔型字段 `append`，若不使用这个字段的话，则其默认值是 `false`。为了打开文件以追加到下一可用的行，可以采用下面的代码：

```
/* 设置FileWriter的append位，保留现有数据，并追加新数据 */
try(BufferedWriter bw = new BufferedWriter(
    new FileWriter("somefile.txt", true))) {
    for(String s : myStringList){
```

```
            bw.write(s);
            /* 不要忘记追加新行! */
            bw.newLine();
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
```

另一种选择是使用 `PrintWriter` 类，它将 `BufferedWriter.FileWriter` 对象作为参数。`PrintWriter` 类有个 `println()` 方法，无论在何种操作系统上，它都采用原生换行符。因此在代码中可以不用 `\n`。这样做的好处是，不用考虑添加那些麻烦的换行符。如果在你自己的计算机（以及相应的操作系统）上生成文本文件，并且是你自己使用这些文件，那么上述方法同样适用。下面是应用 `PrintWriter` 类的例子：

```
try(PrintWriter pw = new PrintWriter(new BufferedWriter(
    new FileWriter("somefile.txt"))) ) {
    for(String s : myStringList){
        /* 添加新行! */
        pw.println(s);
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

对于 JSON 数据，这些方法都适用。采用 `JSONObject.toString()` 方法可以把每个 JSON 对象转换为 `String`，然后把这个 `String` 写入文件中。如果你正在写 JSON 对象，例如配置文件，就像下面一样简单：

```
JSONObject obj = ...

try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    bw.write(obj.toString());
}
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

当创建 JSON 数据文件（关于 JSON 对象的栈）时，遍历 `JSONObjects` 集合：

```
List<JSONObject> dataList = ...

try(BufferedWriter bw = new BufferedWriter(new FileWriter("somefile.txt")) ) {
    for(JSONObject obj : dataList){
        bw.write(obj.toString());
        /* 不要忘记追加新行! */
        bw.newLine();
    }
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

如果文件是要累积的，那么不要忘记设置 FileWriter 中的追加位。只需通过在 FileWriter 中设置追加位，就可以把更多 JSON 记录添加到文件末尾：

```
try(BufferedWriter bw = new BufferedWriter(
    new FileWriter("somefile.txt", true)) ) {
...
}
```

1.5 掌握数据库操作

关系数据库（例如 MySQL）的稳健性与灵活性使得它们成为许多应用场合中最适合的技术。作为一名数据科学家，你很可能会与关系数据库交互，以连接到更大的应用，或者会为数据科学小组的任务生成一些有组织的压缩数据表格。无论哪种情况，掌握命令行、结构化查询语言（SQL）以及 Java 数据库连接（JDBC）都是关键的技能。

1.5.1 命令行客户端

对于管理数据库以及执行查询而言，命令行是很棒的环境。作为交互式 shell，客户端可以对探索数据的命令进行快速修改。查询语句在命令行中通过后，你可以在以后把 SQL 写入到 Java 程序中。为了满足更加灵活的运用，查询中可以包括参数。所有流行的数据库，例如 MySQL、PostgreSQL 和 SQLite，都有命令行客户端。在已经安装了 MySQL（用于开发）的系统中（例如个人计算机），应该能够匿名登录以进行数据库连接，其中数据库名是可选的。

```
bash$ mysql <database>
```

然而，你也许不能创建新数据库。可以用如下命令以数据库管理员身份登录：

```
bash$ mysql -u root <database>
```

这样就具有了完全的访问权限和特权。在其他任何情况下（例如，正连接到生产机器、远程实例，或者基于云的实例），需要下面的方法：

```
bash$ mysql -h host -P port -u user -p password <database>
```

一旦连接上，就可以使用 MySQL shell 进行查询，列出有访问权限的所有数据库、已经连接的数据库名以及用户名：

```
mysql> SHOW DATABASES;
```

若要切换到新数据库，可以采用 USE *dbname* 命令：

```
mysql> USE myDB;
```

现在可以创建表了：

```
mysql> CREATE TABLE my_table(id INT PRIMARY KEY, stuff VARCHAR(256));
```

更进一步，如果把那些创建表的脚本另存为文件，下面的命令将读取并执行该文件：

```
mysql> SOURCE <filename>;
```

当然，也许你想知道数据库中有哪些表。可以用下面的命令：

```
mysql> SHOW TABLES;
```

若想得到表的详细描述，包括列名、数据类型和约束，则可以用下面的命令：

```
mysql> DESCRIBE <tablename>;
```

1.5.2 结构化查询语言

对于浏览数据而言，结构化查询语言（SQL）是个强大的工具。尽管在企业软件应用中，对象关系映射（ORM）框架有一席之地，但是对于数据科学家所面对的任务类型而言，ORM 框架太受限制了。提高 SQL 技能，并先掌握下面的基础知识是个不错的主意。

1. 创建

为了创建数据库和表，采用下面的 SQL 语句：

```
CREATE DATABASE <databasename>;

CREATE TABLE <tablename> (col1 type, col2 type, ...);
```

2. 选择

SELECT 语句的一般形式如下：

```
SELECT
    [DISTINCT]
    col_name, col_name, ... col_name
    FROM table_name
    [WHERE where_condition]
    [GROUP BY col_name [ASC | DESC]]
    [HAVING where_condition]
    [ORDER BY col_name [ASC | DESC]]
    [LIMIT row_count OFFSET offset]
    [INTO OUTFILE 'file_name']
```

有一些技巧迟早可以派上用场。假设数据集包含几百万个数据点，而你只想知道其大致情况，则可以用 ORDER BY 命令返回随机样本：

```
ORDER BY RAND();
```

可以设置 LIMIT 为想要返回的样本大小：

```
ORDER BY RAND() LIMIT 1000;
```

3. 插入

通过下面的语句把数据插入到新行：

```
INSERT INTO tablename(col1, col2, ...) VALUES(val1, val2, ...);
```

注意，如果要插入的是一行的所有列的值，而不是这些列的子集，就可以不给出列名：

```
INSERT INTO tablename VALUES(val1, val2, ...);
```

也可以一次插入多条记录：

```
INSERT INTO tablename(col1, col2, ...) VALUES(val1, val2, ...),(val1, val2, ...),
(val1, val2, ...);
```

4. 更新

在某些情况下，需要更改现有记录。如果需要修补错误或者更正错字，那么大多数时候可以在命令行中快速完成该操作。毫无疑问，数据科学研究人员会访问那些正在从事生产或者正在进行分析与测试的数据库，但也可能正处于临时数据库管理员（DBA）的位置。当涉及实际的用户与数据时，更新记录是很常见的。

```
UPDATE table_name SET col_name = 'value' WHERE other_col_name = 'other_val';
```

在数据科学领域，很难想象需要采用编程的方法来更新数据的场合。当然会有例外，例如前面提到的错字纠正或者逐渐地建立表，但是对于绝大多数情况，更新重要数据听上去像是一场灾难，尤其是当多个用户正依赖于相同数据，并且已经编写代码，随后将针对静态数据集进行分析时。

5. 删除（数据）

如今，存储成本很低，删除数据似乎是不必要的，但是正像 UPDATE，如果产生了错误，却不想重建整个数据库，那么删除就是有用的。通常，你依据特定标准删除记录，例如 user_id、record_id 或者是在某个特定日期之前：

```
DELETE FROM <tablename> WHERE <col_name> = 'col_value';
```

另一个有用的命令是 TRUNCATE，它删除表中所有数据，但是保持表原封不动。本质上，TRUNCATE 的作用是把表擦除干净：

```
TRUNCATE <tablename>;
```

6. 删除（表与数据库）

如果想删除表中所有数据以及表本身，那么必须采用 DROP 命令，该命令彻底删除表：

```
DROP TABLE <tablename>;
```

下面的命令删除整个数据库及其所有内容：

```
DROP DATABASE <databasename>;
```

1.5.3　Java数据库连接

Java 数据库连接（JDBC）是连接 Java 应用与任何支持 SQL 的数据库的协议。每个数据库提供商都有独立的 JAR 文件作为 JDBC 驱动程序，在构建及运行时，必须引入该文件。不论是哪个数据库提供商，JDBC 技术都致力于在各种应用与数据库之间提供统一的接口。

1. 连接

使用 JDBC 连接到数据库是非常简单方便的。只需为数据库填写适当格式的 URI（统一资源标识符），通常的格式如下：

```
String uri = "jdbc:<dbtype>:[location]/<dbname>?<parameters>"
```

`DriverManager.getConnection()` 方法会抛出异常，处理该异常有两种选择。目前 Java 的做法是把连接放在 `try` 语句中，这称为**在 `try` 语句中调用资源**（try with resource）。采用这种方法，执行 `try` 程序段之后，连接会自动关闭，因此不必再显式调用 `Connection.close()` 方法。记住，如果决定把连接语句置入实际的 `try` 程序段中（而不是在 `try` 语句中），那么需要自己显式地关闭连接（很可能是在 `finally` 程序段中）。

```
String uri = "jdbc:mysql://localhost:3306/myDB?user=root";
try(Connection c = DriverManager.getConnection(uri)) {
    // TODO，填写自己的代码
} catch (SQLException e) {
    System.err.println(e.getMessage());
}
```

在拥有了连接之后，有两个问题需弄清楚。

- 在 SQL 字符串中是否有任何变量（是否会以任何方式对 SQL 字符串进行修改）？
- 是否期望从查询中返回任何结果，而不是仅返回查询成功与否的标志？

首先假设将要创建 `Statement` 对象。如果该 `Statement` 对象有变量（例如，将要向 SQL 语句中追加应用变量），那么使用 `PreparedStatement` 对象。若不希望返回任何结果，则这样做就可以了。若希望返回结果，则需要使用 `ResultSets` 对象存放并处理结果。

2. SQL 语句

执行 SQL 语句时，考虑下面的例子：

```
DROP TABLE IF EXISTS data;
CREATE TABLE IF NOT EXISTS data(
```

```
    id INTEGER PRIMARY KEY,
    yr INTEGER,
    city VARCHAR(80));
INSERT INTO data(id, yr, city) VALUES(1, 2015, "San Francisco"),
    (2, 2014, "New York"),(3, 2012, "Los Angeles");
```

本例中所有 SQL 语句都是硬编码的字符串，没有可改变的部分。它们（除了返回布尔类型的编码）没有返回任何值，可以采用下面的方法在上述 `try-catch` 程序段中独立地执行：

```
String sql = "<sql string goes here>";
Statement stmt = c.createStatement();
stmt.execute(sql);
stmt.close();
```

3. 预备语句

也可以不把所有数据硬编码到 SQL 语句中。同样，可以通过 SQL 的 `WHERE` 子句创建通用更新语句，更新给定 `id` 的记录的 `city` 列。尽管你可能禁不住要通过拼接来构造 SQL 字符串，但实际中不推荐这样做。任何时候，若把外部输入替换进 SQL 表达式中，就有可能受到 SQL 注入攻击。合适的方法是在 SQL 语句中采用占位符（即问号），然后使用 `PreparedStatement` 类适当地引用输入变量并执行查询。预备语句的优点不仅包括安全性，也包括速度快。相较于每一次插入操作，都编译一条新 SQL 语句，`PreparedStatement` 只需编译一次。对于大量的插入操作，这种方法可以极大地提高插入过程的效率。前面的 `INSERT` 语句，如果采用相应的 Java 语句，可以写成下面这样：

```
String insertSQL = "INSERT INTO data(id, yr, city) VALUES(?, ?, ?)";
PreparedStatement ps = c.prepareStatement(insertSQL);
/* 设置每个占位符"?"的值，起始索引是1 */
ps.setInt(1, 1);
ps.setInt(2, 2015);
ps.setString(3, "San Francisco");
ps.execute();
ps.close();
```

但如果有大量数据，并且需要遍历列表，那么该如何做呢？可以采用**批处理模式**（batch mode）。例如，假设有一个包含 `Record` 对象的 `List`，其内容是从 CSV 导入的：

```
String insertSQL = "INSERT INTO data(id, yr, city) VALUES(?, ?, ?)";
PreparedStatement ps = c.prepareStatement(insertSQL);
List<Record> records = FileUtils.getRecordsFromCSV();
for(Record r: records) {
    ps.setInt(1, r.id);
    ps.setInt(2, r.year);
    ps.setString(3, r.city);
    ps.addBatch();
}
ps.executeBatch();
ps.close();
```

4. 结果集

SELECT 语句会返回结果。每次编写 SELECT 语句时，都需要调用 Statement.executeQuery() 方法，而不是 execute() 方法，并且要把返回结果赋值给 ResultSet。在数据库术语中，ResultSet 是游标，它是可迭代的数据结构。Java 类 ResultSet 本身实现了 Java 接口 Iterator，从而可以使用熟悉的 while-next 循环。

```
String selectSQL = "SELECT id, yr, city FROM data";
Statement st = c.createStatement();
ResultSet rs = st.executeQuery(selectSQL);
while(rs.next()) {
    int id = rs.getInt("id");
    int year = rs.getInt("yr");
    String city = rs.getString("city"));
    // TODO，对每一行的值进行处理
}
rs.close();
st.close();
```

就像需要对文件逐行进行读取那样，你必须决定如何处理数据。你也许要把每个值存入与值同类型的数组中，或者会把每行数据存入到一个类中，再用该类来建立列表。注意，我们现在正按照数据库模式，根据列名获取列值，进而从 ResultSet 实例中提取值。可以通过从 1 开始递增列索引来实现。

1.6 通过绘图将数据可视化

在数据科学中，数据可视化是令人兴奋的重要组成部分。大量可用的有趣数据与交互式图形技术的结合，产生了令人震惊的可视化效果，将原本复杂的事情讲述得非常清楚。好多时候，可视化是所有人期待的。最重要的是，要意识到，依据所选择展示的数据片段以及所利用的图形样式，同一数据源可以讲出完全不同的事情。

记住，数据可视化必须考虑受众的需要。可视化的消费者大体可以分为三类。首先是你自己，即无所不知的专家，最有可能对数据分析或者算法开发进行快速迭代。你的需要是尽可能简单明了地、快速地看到数据。设置图形的标题、坐标轴标签、平滑、图例，还有日期格式等，这些也许并不重要，因为你自己知道要看的是什么。本质上，我们经常对数据进行绘制，就是为了快速概览数据全貌，并不关心其他人如何查看数据。

数据可视化的第二类消费者是业界专家。解决一个数据科学问题且准备好分享后，关键是要完整地标识坐标轴，为其加上有意义的描述性标题，确保每一系列的数据都用图例做了描述，并且保证所绘制的图本身能大致讲明白一件事情。即使它在视觉上不能给人留下深刻印象，但是同事和同行很可能不在乎那些中看不中用的东西，他们在意的是试图向他们传递的信息。事实上，如果可视化能够做到图形小部件清晰且效果良好的话，那么人们更加容易对一项工作的价值做出科学的评价。当然，这种格式对于数据归档也是必要的。若

现在不对坐标轴进行标记，一个月后，你将记不得它们代表什么了。

数据可视化的第三类消费者是其他所有人。现在是发挥创造性以及艺术性的时候了，因为仔细选择颜色和样式可以使好的数据显得更棒。然而要小心的是，你需要花费大量的时间以及精力来为这个层次的消费者准备图形。采用 JavaFX 的额外好处是，通过鼠标选项可以实现交互性。这使得你可以构造出图形应用，它类似于人们所习惯的众多基于 Web 的仪表盘。

1.6.1 创建简单图形

在 JavaFX 包中，Java 自带图形能力。从 1.8 版本起，借助于 `javafx.scene.chart` 包，可以用多种类型的图表进行科学绘图，例如散点图、折线图、条形图、堆叠条形图、饼图、面积图、堆叠面积图或者气泡图。`Stage` 对象包含 `Scene` 对象，`Scene` 对象包含 `Chart` 对象。一般形式是采用 `Application` 扩展可执行的 Java 类，并把所有绘制命令放在重写的 `Application.start()` 方法中。必须在 `main` 方法中调用 `Application.launch()` 方法，以创建并显示图表。

1. 散点图的绘制

简单绘图的一个例子是散点图，它把一系列 *x*-*y* 对的数字绘制成网格上的点。这些图利用了 `javafx.scene.chart.XYChart.Data` 与 `javafx.scene.chart.XYChart.Series` 两个类。`Data` 类是容器，它可容纳任何规模的混合类型的数据，`Series` 类包含内容为 `Data` 实例的 `ObservableList`。在 `javafx.collections.FXCollections` 类中有一些工厂方法，如果愿意的话，可以用这些方法直接创建 `ObservableList` 的实例。不过，对于散点图、折线图、面积图、气泡图以及条形图，这是不必要的，因为它们都利用了 `Series` 类。

```
public class BasicScatterChart extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        int[] xData = {1, 2, 3, 4, 5};
        double[] yData = {1.3, 2.1, 3.3, 4.0, 4.8};

        /* 将Data添加到Series中 */
        Series series = new Series();
        for (int i = 0; i < xData.length; i++) {
            series.getData().add(new Data(xData[i], yData[i]));
        }

        /* 定义坐标轴 */
        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("x");
```

```
        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("y");

        /* 创建散点图 */
        ScatterChart<Number,Number> scatterChart =
            new ScatterChart<>(xAxis, yAxis);
        scatterChart.getData().add(series);

        /* 采用图创建场景 */
        Scene scene = new Scene(scatterChart, 800, 600);

        /* 告诉舞台（stage）采用什么场景（scene）并对其进行绘制 */
        stage.setScene(scene);
        stage.show();
    }

}
```

图 1-1 是用 JavaFX 绘制简单数据集后的默认图形窗口。

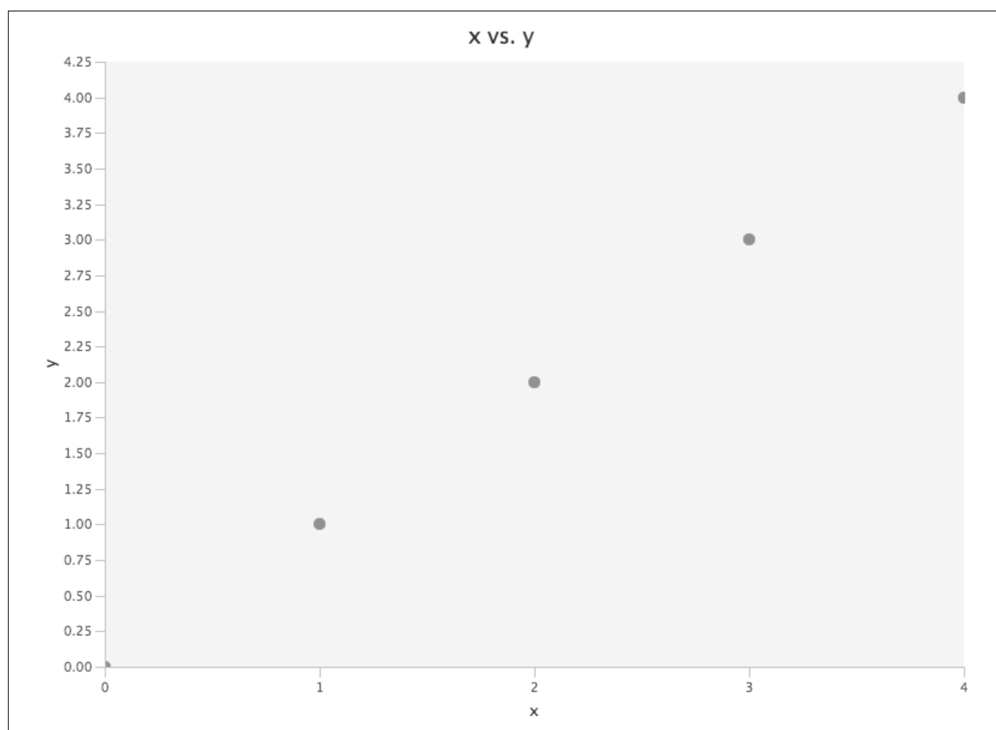


图 1-1：散点图示例

在前面的例子中，可以方便地采用 LineChart、AreaChart 或者 BubbleChart 类替换 ScatterChart 类。

2. 条形图

作为 x-y 图，条形图利用了 Data 与 Series 类。然而，在这种情况下，唯一的区别是，x 轴必须是字符串类型（而不是数值类型），并且利用 CategoryAxis 类，而不是 NumberAxis 类。y 轴仍然利用 NumberAxis 类。通常，条形图的分类就像一周中的每一天，或者市场的细分。注意，BarChart 类内部采用的是 (String, Number) 对的类型，这对生成直方图是有用的（第 3 章会展示一个直方图）。

```
public class BasicBarChart extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) throws Exception {

        String[] xData = {"Mon", "Tues", "Wed", "Thurs", "Fri"};
        double[] yData = {1.3, 2.1, 3.3, 4.0, 4.8};

        /* 把Data添加到Series中 */
        Series series = new Series();
        for (int i = 0; i < xData.length; i++) {
            series.getData().add(new Data(xData[i], yData[i]));
        }

        /* 定义坐标 */
        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("x");
        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("y");

        /* 创建条形图 */
        BarChart<String,Number> barChart = new barChart<>(xAxis, yAxis);
        barChart.getData().add(series);


        /* 采用图创建场景 */
        Scene scene = new Scene(barChart, 800, 600);

        /* 告诉舞台（stage）采用什么场景（scene）并对其进行绘制 */
        stage.setScene(scene);
        stage.show();
    }

}
```

3. 绘制多个系列

可以很容易地实现对任何图形类型的多个系列进行绘制。对于散点图的示例，只需要创建多个 Series 实例：

```
Series series1 = new Series();
Series series2 = new Series();
Series series3 = new Series();
```

然后，用 addAll() 方法，而不是用 add() 方法，可以一次添加所有这些系列：

```
scatterChart.getData().addAll(series1, series2, series3);
```

从所产生的图中可以看出，这些点以各种颜色重叠在一起，每一种颜色以各自的图例来指示其标签名。这对于折线图、面积图、条形图、气泡图同样适用。对于 StackedAreaChart 与 StackedBarChart 这两个类，它们的一个特点值得我们关注，那就是除了一个数据是堆叠在另一个数据之上的，以免在视觉上相互遮挡外，它们运行的方式与其各自的超类 AreaChart 和 BarChart 相同。

当然，有时候，采用多种类型的图对数据进行混合绘制，可视化效果会更好，例如对同一数据同时采用散点图与折线图。当前，Scene 类只接受同一种类型的图表。不过，本章随后将给出一些变通方法。

4. 基本格式

有一些有用的选项，可以使图看起来更专业。首先需要清理的地方可能是坐标轴。通常，刻度线太小会适得其反。也可以通过设置最小值与最大值来界定图的范围。

```
scatterChart.setBackground(null);
scatterChart.setLegendVisible(false);
scatterChart.setHorizontalGridLinesVisible(false);
scatterChart.setVerticalGridLinesVisible(false);
scatterChart.setVerticalZeroLineVisible(false);
```

在某个阶段，保持绘制方法简洁，并把所有样式说明包含在 CSS（层叠样式表）文件中，也许会更容易一些。JavaFX 8 中默认的 CSS 称作 Modena，若不修改样式选项，则会应用 Modena 文件。也可以创建自己的 CSS，并采用下面的方法将其包含在场景中：

```
scene.getStylesheets().add("chart.css");
```

默认路径是自己 Java 包的 src/main/resources 目录。

1.6.2 混合类型图的绘制

我们经常想要在同一个图形中显示多个不同类型的图。例如，有时需要把数据点显示为 *x-y* 散点图，然后在散点图上覆盖一个采用最佳拟合模型产生的折线图。也许还想在图中包含另外两条折线，用以表示模型的边界，可能是一倍、两倍或三倍的标准差 σ，也可能是置信区间 $1.96 \times \sigma$。目前，JavaFX 不允许在同一场景中同时显示多个不同类型的图。不过，有个变通方法：可以用 LineChart 类绘制多个系列的 LineChart 实例，然后用 CSS 设置其样式为：第一条折线只显示点，第二条折线只显示实线，再有两条折线只显示虚线。

CSS 如下：

```
.default-color0.chart-series-line {
    -fx-stroke: transparent;
}

.default-color1.chart-series-line {
    -fx-stroke: blue; -fx-stroke-width: 1;
}

.default-color2.chart-series-line {
    -fx-stroke: blue;
    -fx-stroke-width: 1;
    -fx-stroke-dash-array: 1 4 1 4;
}

.default-color3.chart-series-line {
    -fx-stroke: blue;
    -fx-stroke-width: 1;
    -fx-stroke-dash-array: 1 4 1 4;
}

/*.default-color0.chart-line-symbol {
    -fx-background-color: white, green;
}*/

.default-color1.chart-line-symbol {
    -fx-background-color: transparent, transparent;
}

.default-color2.chart-line-symbol {
    -fx-background-color: transparent, transparent;
}

.default-color3.chart-line-symbol {
    -fx-background-color: transparent, transparent;
}
```

绘图如图 1-2 所示。

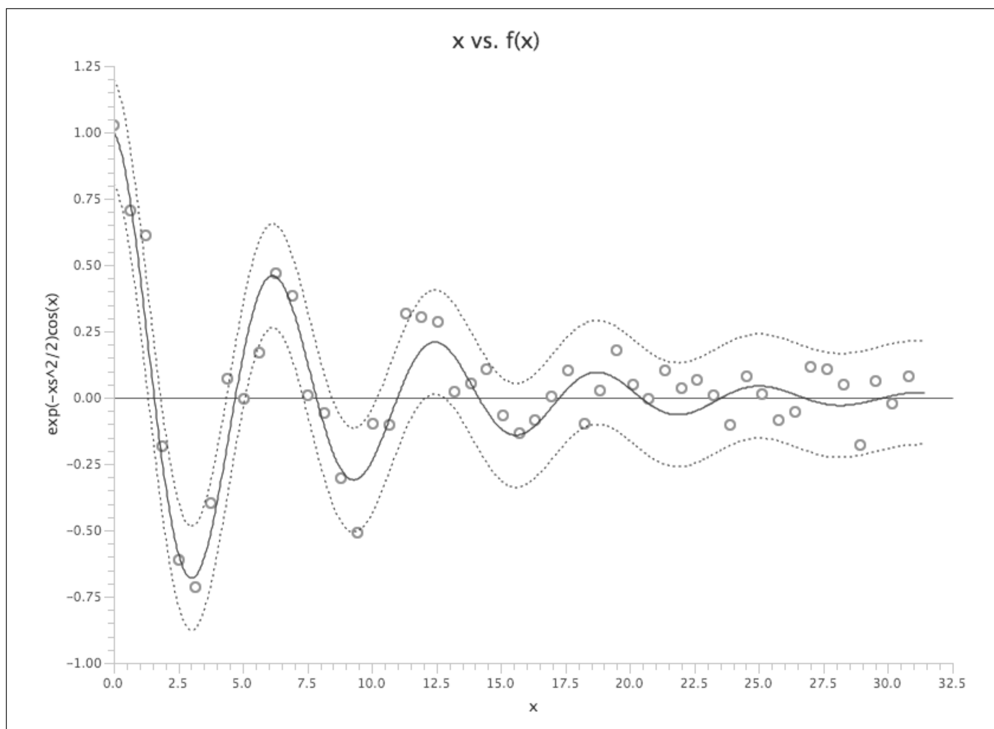


图 1-2：采用 CSS 实现混合折线类型的绘制

1.6.3 把图存入文件

毫无疑问，有时候需要把图存入文件中。你也许会在电子邮件中发送该图，或者把它包含在演示文稿中。综合采用标准 Java 类与 JavaFX 类，可以很容易地以任何格式保存图。采用 CSS，甚至可以使所绘制的图成为具有出版质量的图形。事实上，本章（以及书中其他部分）的图就是用这种方式生成的。

每一种图表类都是抽象类 `Chart` 的子类，`Chart` 从 `Node` 类中继承了 `snapshot()` 方法。`Chart.snapshot()` 方法返回 `WritableImage` 类。有个潜在的不利因素需要指出：采用场景将数据绘制到图表上时，保存图的文件中将不会有该图上的实际数据。在实例化图表之后，在采用 `Chart.getData.add()` 或其他等效方法把数据添加到图表之前，通过 `Chart.setAnimated(false)` 关掉动画是极其重要的。

```
/* 把图表实例化之后，立刻进行该操作 */
scatterChart.setAnimated(false);
...
/* 绘制图片 */
stage.show();
...
```

```
/* 绘制舞台之后，再把图保存到文件中 */
WritableImage image = scatterChart.snapshot(new SnapshotParameters(), null);
File file = new File("chart.png");
ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png", file);
```



本书中的所有数据图都是采用 JavaFX 8 绘制的。

第 2 章

线性代数

前面已经用一整章的篇幅讨论了以某种格式获取数据。我们很可能要以电子表格的形式查看数据。我们自然可以预想一下表格中每一列的名称，例如从左向右分别是年龄、地址、ID 号等。表格中的每一行代表一条唯一的记录或数据点。数据科学中的许多数据就是以这种格式呈现的。我们所要寻找的是，想要关注的任意数量的列［称为**变量**（variable）］与表示可度量结果的任意数量的行［称为**响应**（response）］之间的关系。

通常，字母 x 表示变量，字母 y 表示响应。同样，响应可以用矩阵 $\boldsymbol{Y}$ 表示，它的列数为 p，并且 p 必须与矩阵 $\boldsymbol{X}$ 的行数 m 相同。注意，在很多情况下，只有一维的响应变量，即 $p = 1$。不过，可以把线性代数问题推广到任意维。

一般来说，线性代数背后的主要思想是寻找 $\boldsymbol{X}$ 与 $\boldsymbol{Y}$ 之间的关系。这些关系之中最简单的情形是，是否可以用新的待定值矩阵 $\boldsymbol{W}$ 去乘 $\boldsymbol{X}$，且使结果准确地（或近似地）等于 $\boldsymbol{Y}$。下面是 $\boldsymbol{XW} = \boldsymbol{Y}$ 的例子。

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,p} \end{pmatrix}$$

注意，等式中给出的矩阵大小看上去差不多，这会造成误导，因为大多数情况下，数据点的数目 m 很大，可能是几百万或几十亿，而矩阵 $\boldsymbol{X}$ 与矩阵 $\boldsymbol{Y}$ 各自的列数 n、p 通常很小（几十到几百）。接下来需要引起注意的是，无论 m 有多大（例如 100 000），矩阵 $\boldsymbol{W}$ 的大小都与 m 的大小无关，它的大小是 $n \times p$（例如 10×10）。这就是线性代数的核心，我们

可以采用紧凑得多的数据结构 $\boldsymbol{W}$ 来解释非常大的数据结构的内容，例如 $\boldsymbol{X}$ 与 $\boldsymbol{Y}$。线性代数的规则使得我们可以根据 $\boldsymbol{X}$ 的一行与 $\boldsymbol{W}$ 的一列来表示 $\boldsymbol{Y}$ 中任意特定的值。例如，$y_{1,1}$ 的值可以用下式表示：

$$y_{1,1} = x_{1,1}\omega_{1,1} + x_{1,2}\omega_{2,1} + \cdots + x_{1,n}\omega_{n,1}$$

本章余下内容将讨论线性代数的规则与运算，最后一节会给出线性系统 $\boldsymbol{XW} = \boldsymbol{Y}$ 的解。数据科学中更高级的话题，比如第 4 章与第 5 章给出的那些，在很大程度上依赖于线性代数。

2.1 构造向量和矩阵

不论采用哪种形式的定义，**向量**（vector）就是给定维数[1]的一维数组[2]，可以举出很多例子。比如整型数组，表示 Web 指标每天的值。再比如存在大量“特征”的数组，该数组将作为机器学习例程的输入。还有对几何坐标的跟踪，例如 x 与 y，可以为每个 `[x, y]` 对创建数组。虽然可以从哲学角度讨论向量（即向量空间中具有大小与方向的元素）的实际含义是什么，但是只要在解决问题的过程中，对向量的定义保持一致，就可以很好地使用所有数学公式，不需要关心它的哲学意义。

一般来说，向量 $\boldsymbol{x}$ 具有以下的形式，它由 n 个分量构成。

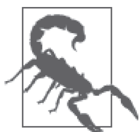
$$\boldsymbol{x} = (x_1 \; x_2 \; \cdots \; x_n)$$

同样，矩阵 $\boldsymbol{A}$ 就是具有 m 行与 n 列的二维数组。

$$\boldsymbol{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

也可以采用矩阵表示法，把向量表示为列向量。

$$\boldsymbol{X} = \begin{pmatrix} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{m,1} \end{pmatrix}$$



本书中小写黑斜体字母表示向量，大写黑斜体字母表示矩阵。注意，向量 $\boldsymbol{x}$ 也可以表示为矩阵 $\boldsymbol{X}$ 的一个列向量。

注 1：此处的维数，是从线性代数角度讨论向量时，习惯用的说法。——译者注

注 2：此处指的一维数组，是从数据结构、程序设计角度来讨论的，与向量的维数是不同的概念。——译者注

在实际中，向量与矩阵对数据科学家来说都是有用的。一个常见的例子是，数据集的（特征）向量是相互叠加的，并且行数 m 要远大于列数 n。本质上，这种数据结构就是一系列向量，只是把它们以矩阵形式处理，以便于对各种线性代数的量进行有效计算。数据科学中遇到的另一类矩阵，其元素代表变量之间的关系，例如协方差矩阵或者相关矩阵。

2.1.1 数组存储

Apache Commons Math 库提供了使用 `RealVector` 类与 `RealMatrix` 类来创建实数向量与实数矩阵的几种选项。3 种最有用的构造方法分别列举如下：分配已知维数的空实例；根据值数组来创建实例；通过深度复制现存的实例来创建另一个实例。为了实例化 `RealVector` 类型的 n 维空向量，可以采用具有整数维数的 `ArrayRealVector` 类。

```
int size = 3;
RealVector vector = new ArrayRealVector(size);
```

如果已经有了值数组，那么可以通过把该数组作为构造方法的参数来创建向量。

```
double[] data = {1.0, 2.2, 4.5};
RealVector vector = new ArrayRealVector(data);
```

可以对已有向量进行深度复制，从而创建新向量的实例[3]。

```
RealVector vector = new ArrayRealVector(realVector);
```

若要把向量的所有分量都设置为默认值，可以在构造方法中给出该值，同时给出向量的大小。

```
int size = 3;
double defaultValue = 1.0;
RealVector vector = new ArrayRealVector(size, defaultValue);
```

对于矩阵实例化，后面也有一些类似的构造方法。可以采用下面的方法对已知大小的空矩阵进行实例化：

```
int rowDimension = 10;
int colDimension = 20;
RealMatrix matrix = new Array2DRowRealMatrix(rowDimension, colDimension);
```

或者，如果已经有二维数组，其中值为 `double` 型，那么可以把该数组作为参数传递给构造方法。

```
double[][] data = {{1.0, 2.2, 3.3}, {2.2, 6.2, 6.3}, {3.3, 6.3, 5.1}};
RealMatrix matrix = new Array2DRowRealMatrix(data);
```

注 3：新向量的元素与已有向量对应位置的元素相同。——译者注

尽管没有方法把整个矩阵设置为默认值（就像向量一样），但是可以先实例化新矩阵，并把所有元素设置为 0，这样随后就可以很容易地让每个元素加上同一值。

```
int rowDimension = 10;
int colDimension = 20;
double defaultValue = 1.0;
RealMatrix matrix = new Array2DRowRealMatrix(rowDimension, colDimension);
matrix.scalarAdd(defaultValue);
```

可以采用 RealMatrix.copy() 方法对矩阵进行深度复制。

```
/* 深度复制矩阵的内容 */
RealMatrix anotherMatrix = matrix.copy();
```

2.1.2　块存储

对于维数大于 50 的大矩阵，推荐采用 BlockRealMatrix 类所实现的块存储。块存储是前一节讨论的二维数组存储的替代方法。在这种情况下，大矩阵可以划分为一些较小的数据块，从而更加便于缓存和处理。为了给矩阵分配空间，可以采用下面的构造方法：

```
RealMatrix blockMatrix = new BlockRealMatrix(50, 50);
```

或者，如果已经有二维数组的数据，那么可以使用下面的构造方法[4]：

```
double[][] data = ;
RealMatrix blockMatrix = new BlockRealMatrix(data);
```

2.1.3　映射存储

如果有大向量或矩阵，其内容几乎全是零，则称为**稀疏的**（sparse）。因为存储所有零是低效的，所以仅存储那些非零分量（或元素）的位置以及值。事实上，在 HashMap 中可以很容易地存储这些值。要创建已知维数的稀疏向量，可以采用下面的方法：

```
int dim = 10000;
RealVector sparseVector = new OpenMapRealVector(dim);
```

要创建稀疏矩阵，只需要再加上另一维即可。

```
int rows = 10000;
int cols = 10000;
RealMatrix sparseMatrix = new OpenMapRealMatrix(rows, cols);
```

2.1.4　访问元素

不论底层采取何种方法存储向量与矩阵，对其赋值以及随后获取值的方法都是等效的。

注 4：以下代码在实际使用时应当给出二维数组的值。——译者注



虽然在本书给出的线性代数理论中，索引是从 1 开始的，但是在 Java 语言中索引是从 0 开始的。把算法从理论转变为代码时，要牢记这一点，尤其是在设定值与获取值时。

采用 `setEntry(int index, double value)` 方法以及 `getEntry(int index)` 方法分别设定值与获取值。

```
/* 设定向量v的第一个值 */
vector.setEntry(0, 1.2)
/* 再获取该值 */
double val = vector.getEntry(0);
```

要把向量的所有值设置为相同值，采用 `set(double value)` 方法。

```
/* 把向量的所有值置为0 */
vector.set(0);
```

然而，如果 ***v*** 是稀疏向量，则没有必要一次设定所有值。在稀疏代数中，缺失值默认设定为 0。只要使用 `setEntry` 方法设置那些非零值即可。如果要获取已有向量中的所有值，并把它作为 `double` 型数组，那么可采用 `toArray()` 方法。

```
double[] vals = vector.toArray();
```

不论采用何种存储方法，都同样存在针对矩阵设定值与获取值的方法。可以分别采用 `setEntry(int row, int column, double value)` 以及 `getEntry(int row, int column)` 方法如下：

```
/* 设置第一行、第三列的元素为3.14 */
matrix.setEntry(0, 2, 3.14);
/* 再获取该元素 */
double val = matrix.getEntry(0, 2);
```

与向量的类不同，没有 `set()` 方法可以把矩阵所有元素设置为相同值。不过，就像新构造矩阵那样，只要矩阵的所有元素都设置为 0，就可以通过加常量的方法把矩阵所有元素设定为相同值，代码如下：

```
/* 对于已经存在的新矩阵 */
matrix.scalarAdd(defaultValue);
```

正如稀疏向量那样，把稀疏矩阵的每个 i, j 对所对应的元素全部设置为 0 是没有用处的。

为了以 `double` 型数组的形式获得矩阵的所有值，可以采用 `getData()` 方法。

```
double[][] matrixData = matrix.getData();
```

2.1.5 处理子阵

我们经常只需要处理矩阵的特定部分，或者想把小一些的矩阵包含在大一些的矩阵之中。RealMatrix 类有几个实用的方法来应对这些常见情况。有两种方法为已有矩阵创建它的子阵。第一种方法是从源矩阵中选择矩形区域，并用这些项创建新矩阵。选定的矩形区域是由分别位于源矩阵左上角和所包含区域右下角的两个起点所确定的，它们都包含在选定区域内。这可以通过调用 RealMatrix.getSubMatrix(int startRow, int endRow, int startColumn, int endColumn) 方法来实现，返回 RealMatrix 对象，新矩阵（子阵）的维数和值由选定的区域确定。注意，endRow 与 endColumn 的值包含在选定区域之内。

```
double[][] data = {{1,2,3},{4,5,6},{7,8,9}};
RealMatrix m = new Array2DRowRealMatrix(data);
int startRow = 0;
int endRow = 1;
int startColumn = 1;
int endColumn = 2;
RealMatrix subM = m.getSubMatrix(startRow, endRow, startColumn, endColumn);
// {{2,3},{5,6}}
```

我们也可以获得矩阵的指定行与指定列。为此，可以为行与列分别创建整型数组，其元素表示想要保留的行与列的索引。然后，在 RealMatrix.getSubMatrix(int[] selectedRows, int[] selectedColumns) 方法中把这两个数组作为参数，于是就有了下面 3 种应用场景：

```
/* 获得选定的行与所有列 */
int[] selectedRows = {0, 2};
int[] selectedCols = {0, 1, 2};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{1,2,3},{7,8,9}}

/* 获得所有的行与选定的列 */
int[] selectedRows = {0, 1, 2};
int[] selectedCols = {0, 2};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{1,3},{4,6},{7,9}}

/* 获得选定的行与选定的列 */
int[] selectedRows = {0, 2};
int[] selectedCols = {1};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{2},{8}}
```

也可以通过设置子阵的值逐步地创建矩阵。这可以通过在 RealMatrix.setSubMatrix(double[][] subMatrix, int row, int column) 方法中传入行与列的位置，并向已有矩阵添加 double 型数据数组来实现。

```
double[][] newData = {{-3, -2}, {-1, 0}};
int row = 0;
int column = 0;
m.setSubMatrix(newData, row, column);
// {{-3,-2,3},{-1,0,6},{7,8,9}}
```

2.1.6 随机化

在机器学习算法中，经常需要把矩阵（或向量）的所有值设置为随机数。可以选择实现了 AbstractRealDistribution 接口的统计分布，或者直接采用简易的构造方法，选择 −1~1 范围内的随机数。可以把这些值传递给已有矩阵或者向量，也可以创建新实例。

```
public class RandomizedMatrix {

    private AbstractRealDistribution distribution;

    public RandomizedMatrix(AbstractRealDistribution distribution, long seed) {
        this.distribution = distribution;
        distribution.reseedRandomGenerator(seed);
    }

    public RandomizedMatrix() {
        this(new UniformRealDistribution(-1, 1), 0L);
    }

    public void fillMatrix(RealMatrix matrix) {
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            matrix.setRow(i, distribution.sample(matrix.getColumnDimension()));
        }
    }

    public RealMatrix getMatrix(int numRows, int numCols) {
        RealMatrix output = new BlockRealMatrix(numRows, numCols);
        for (int i = 0; i < numRows; i++) {
            output.setRow(i, distribution.sample(numCols));
        }
        return output;
    }

    public void fillVector(RealVector vector) {
        for (int i = 0; i < vector.getDimension(); i++) {
            vector.setEntry(i, distribution.sample());
        }
    }

    public RealVector getVector(int dim) {
        return new ArrayRealVector(distribution.sample(dim));
    }
}
```

可以采用下面的方法创建元素值服从正态分布的窄带（narrow band）：

```
int numRows = 3;
int numCols = 4;
long seed = 0L;
RandomizedMatrix rndMatrix = new RandomizedMatrix(
    new NormalDistribution(0.0, 0.5), seed);
RealMatrix matrix = rndMatrix.getMatrix(numRows, numCols);
```

```
// -0.0217405716,-0.5116704988,-0.3545966969,0.4406692276
// 0.5230193567,-0.7567264361,-0.5376075694,-0.1607391808
// 0.3181005362,0.6719107279,0.2390245133,-0.1227799426
```

2.2　向量与矩阵的运算

你有时对自己探索的算法或数据结构有一些构想，但是不能确定如何实现。可以在头脑中做一些“智力模式匹配”，然后选择实现（例如，是做点积，而不是自己手动遍历所有数据）。下面就来探讨线性代数中的一些常见运算。

2.2.1　缩放

用常数 κ 对向量缩放（相乘）的运算如下：

$$\kappa \boldsymbol{x} = (\kappa x_1, \kappa x_2, \cdots, \kappa x_n)$$

Apache Commons Math 实现了映射方法，可以用已有 `RealVector` 对象乘以 `double` 型数来生成新 `RealVector` 对象。

```
double k = 1.2;
RealVector scaledVector = vector.mapMultiply(k);
```

注意 `RealVector` 对象也可以就地缩放，方法是永久修改已有向量。

```
vector.mapMultiplyToSelf(k);
```

同样，可以用向量除以 κ 来创建新向量。

```
RealVector scaledVector = vector.mapDivide(k);
```

以下是就地除法：

```
vector.mapDivideToSelf(k);
```

矩阵 $\boldsymbol{A}$ 也可以用数 κ 来缩放。

$$\kappa \boldsymbol{A} = \begin{pmatrix} \kappa a_{1,1} & \kappa a_{1,2} & \cdots & \kappa a_{1,n} \\ \kappa a_{2,1} & \kappa a_{2,2} & \cdots & \kappa a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \kappa a_{m,1} & \kappa a_{m,2} & \cdots & \kappa a_{m,n} \end{pmatrix}$$

此处矩阵的每个值都乘以 `double` 型的常数，从而返回新矩阵。

```
double k = 1.2;
RealMatrix scaledMatrix = matrix.scalarMultiply(k);
```

2.2.2 转置

对矩阵或向量进行**转置**（transpose），类似于把它沿着从左上角到右下角的对角线翻转过来。向量 $\boldsymbol{x}$ 的转置记作 $\boldsymbol{x}^{\mathrm{T}}$，矩阵 $\boldsymbol{A}$ 的转置记作 $\boldsymbol{A}^{\mathrm{T}}$。许多情况下没有必要计算向量的转置，因为在实现时，`RealVector` 类与 `RealMatrix` 类的方法已考虑了向量转置的需要。除非向量以矩阵形式表示，否则其转置是未定义的。于是，$m \times 1$ 列向量的转置是新的大小为 $1 \times m$ 的行向量矩阵。

$$\boldsymbol{x}^{\mathrm{T}} = (x_1, x_2, \cdots, x_m)$$

若确实需要对向量进行转置，则可以直接把数据插入到 `RealMatrix` 实例中。采用元素值为 `double` 型的一维数组作为参数，提供给 `Array2DRowRealMatrix` 类，可以创建 m 行 1 列的矩阵，其值由 `double` 型数组提供。把列向量进行转置将返回 1 行 m 列的矩阵。

```
double[] data = {1.2, 3.4, 5.6};
RealMatrix columnVector = new Array2DRowRealMatrix(data);
System.out.println(columnVector);
/* {{1.2}, {3.4}, {5.6}} */
RealMatrix rowVector = columnVector.transpose();
System.out.println(rowVector);
/* {{1.2, 3.4, 5.6}} */
```

对 $m \times n$ 矩阵进行转置，结果是 $n \times m$ 矩阵。简单来说，就是把行与列的索引 i 与 j 互相交换。

$$\boldsymbol{A}^{\mathrm{T}} = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{pmatrix}$$

注意，矩阵转置运算将返回新矩阵。

```
double[][] data = {{1, 2, 3}, {4, 5, 6}};
RealMatrix matrix = new Array2DRowRealMatrix(data);
RealMatrix transposedMatrix = matrix.transpose();
/* {{1, 4}, {2, 5}, {3, 6}} */
```

2.2.3 加与减

两个维数都为 n 的向量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ **相加**，产生了维数为 n 的向量，其值等于两个向量索引相同的分量进行相加。

$$\boldsymbol{a} + \boldsymbol{b} = (a_1 + b_1, a_2 + b_2, \cdots, a_n + b_n)$$

其结果是新的 `RealVector` 实例：

```
RealVector aPlusB = vectorA.add(vectorB);
```

同样，两个维数为 n 的 RealVector 对象**相减**结果如下：

$$\boldsymbol{a} - \boldsymbol{b} = (a_1 - b_1, a_2 - b_2, \cdots, a_n - b_n)$$

这会返回新的 RealVector 对象，其值是两个向量索引相同的分量相减。

```
RealVector aMinusB = vectorA.subtract(vectorB);
```

类似于向量，相同大小的矩阵也可以进行加减运算。

$$\boldsymbol{A} + \boldsymbol{B} = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \cdots & a_{1,n} + b_{1,n} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \cdots & a_{2,n} + b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} + b_{m,1} & a_{m,2} + b_{m,2} & \cdots & a_{m,n} + b_{m,n} \end{pmatrix}$$

RealMatrix 对象 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 进行加减运算，返回新的 RealMatrix 实例。

```
RealMatrix aPlusB = matrixA.add(matrixB);
RealMatrix aMinusB = matrixA.subtract(matrixB);
```

2.2.4　长度

向量**长度**（length）是把向量所有分量归纳为一个数值的一种简便方法，但是不要与向量维数混淆。现在有几种向量长度的定义，最常用的两种是 L1 范数与 L2 范数。其中，L1 范数可用于确保概率向量或者某些分数的所有值总和为 1。

$$|\boldsymbol{x}| = \sum_{i=1}^{n} |x_i|$$

L1 范数不如 L2 范数常用，因此用全名“L1 范数”来表示，以避免歧义。

```
double norm = vector.getL1Norm();
```

L2 范数通常用于对向量归一化，大多数时候被称作**范数**（norm）或者向量**大小**（magnitude），其数学定义如下：

$$\|\boldsymbol{x}\| = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

```
/* 计算向量的L2范数 */
double norm = vector.getNorm();
```



人们经常问何时使用 L1 向量长度或 L2 向量长度。实际中，这取决于向量代表什么。在一些场合中，需要将计数或者概率收集到向量中。此时，需要把每个分量除以各个分量的总和，以对向量进行归一化（L1）。另一方面，若向量包含某种坐标或者特征，则会用向量的欧氏距离对其进行归一化（L2）。

向量与其对应的**单位向量**（unit vector）所指方向相同。因为已经通过 L2 范数将其长度缩放至 1，所以它称为单位向量，通常记作 $\hat{x}$，计算公式如下：

$$\hat{x} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}$$

`RealVector.unitVector()` 方法返回新的 `RealVector` 对象。

```
/* 创建新向量，它是某向量实例的单位向量 */
RealVector unitVector = vector.unitVector();
```

向量还可以通过就地缩放成为单位向量。向量 $\boldsymbol{v}$ 可以通过下面的方法永久地转换成自己的单位向量：

```
/* 把向量就地转换为单位向量 */
vector.unitize();
```

也可以通过 Frobenius 范数计算矩阵的范数。Frobenius 范数在数学上表示为所有元素平方和的平方根。

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

在 Java 中，这可以通过下面的方法得到：

```
double matrixNorm = matrix.getFrobeniusNorm();
```

2.2.5 距离

可以用几种方式计算任意两个向量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 之间的**距离**。向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 之间 L1 距离的表达式如下：

$$d_{\mathrm{L1}} = \sum_{i=1}^{n}|a_i - b_i|$$

```
double l1Distance = vectorA.getL1Distance(vectorB);
```

L2 距离（也称作欧氏距离）的表达式如下：

$$d_{L2} = \sqrt{\sum_{i=1}^{n} |a_i - b_i|^2}$$

向量之间的距离较为常见的计算方式是 L2 距离。`Vector.getDistance(RealVector vector)` 方法可以返回欧氏距离。

```
double l2Distance = vectorA.getDistance(vectorB);
```

余弦距离（cosine distance）是位于 −1~1 范围内的度量，它与其说是距离指标，不如说是“相似性”度量。若 $d = 0$，则两个向量是正交的（没有任何共同点）。若 $d = 1$，则两个向量指向同一方向。若 $d = -1$，则两个向量指向完全相反的方向。余弦距离也可以看作两个单位向量的点积。

$$d = \cos(\theta) = \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{\|\boldsymbol{a}\| \, \|\boldsymbol{b}\|}$$

```
double cosineDistance = vectorA.cosine(vectorB);
```

若 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 是单位向量，则余弦距离恰好就是它们的内积。

$$d = \cos(\theta) = \hat{\boldsymbol{a}} \cdot \hat{\boldsymbol{b}}$$

`Vector.dotProduct(RealVector vector)` 方法可以计算点积。

```
/* 对于单位向量a与b */
vectorA.unitize();
vectorB.unitize();
double cosineDistance = vectorA.dotProduct(vectorB);
```

2.2.6 相乘

$m \times n$ 矩阵 $\boldsymbol{A}$ 与 $n \times p$ 矩阵 $\boldsymbol{B}$ 相乘，得到 $m \times p$ 矩阵。其中，唯一必须匹配的维数是 n，也就是 $\boldsymbol{A}$ 的列数与 $\boldsymbol{B}$ 的行数。

$$\boldsymbol{AB} = \begin{pmatrix} (AB)_{1,1} & (AB)_{1,2} & \cdots & (AB)_{1,p} \\ (AB)_{2,1} & (AB)_{2,2} & \cdots & (AB)_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{m,1} & (AB)_{m,2} & \cdots & (AB)_{m,p} \end{pmatrix}$$

元素 $(AB)_{ij}$ 的值是 $\boldsymbol{A}$ 的第 i 行每个元素与 $\boldsymbol{B}$ 的第 j 列每个元素的乘积之和，数学表达式如下：

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

矩阵 $\boldsymbol{A}$ 乘以矩阵 $\boldsymbol{B}$ 的结果可以由下面的代码得到：

```
RealMatrix matrixMatrixProduct = matrixA.multiply(matrixB);
```

注意 $\boldsymbol{AB} \neq \boldsymbol{BA}$。$\boldsymbol{BA}$ 可以显式地计算，也可以采用左乘（preMultiply）方法，两种方法结果相同。然而在这种情形下，注意 $\boldsymbol{B}$ 的列数必须与 $\boldsymbol{A}$ 的行数相同。

```
/* 显式计算BA */
RealMatrix matrixMatrixProduct matrixB.multiply(matrixA);

/* 采用左乘（preMultiply）方法计算BA */
RealMatrix matrixMatrixProduct = matrixA.preMultiply(matrixB);
```

矩阵相乘也常用于 $m \times n$ 矩阵 $\boldsymbol{A}$ 与 $n \times 1$ 列向量 $\boldsymbol{x}$ 的相乘，结果是 $m \times 1$ 列向量 $\boldsymbol{b}$，即 $\boldsymbol{Ax} = \boldsymbol{b}$。通过把 $\boldsymbol{A}$ 矩阵第 i 行的每个元素与向量 $\boldsymbol{x}$ 的每个元素相乘，然后求和，可以完成该运算，矩阵表示法如下：

$$\boldsymbol{Ax} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \end{pmatrix}$$

下面的代码与前面的矩阵与矩阵相乘相同。

```
/* Ax的结果是列向量 */
RealMatrix matrixVectorProduct = matrixA.multiply(columnVectorX);
```

我们经常需要计算向量与矩阵的乘积，通常记作 $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}$。当 $\boldsymbol{x}$ 是矩阵形式时，可以按下面的方法显式地完成该计算。

```
/* 显式计算xTA */
RealMatrix vectorMatrixProduct = columnVectorX.transpose().multiply(matrixA);
```

若 $\boldsymbol{x}$ 是 RealVector 对象，则可以用 RealMatrix.preMultiply() 方法来计算。

```
/* 采用左乘方法（preMultiply）计算xTA */
RealMatrix vectorMatrixProduct = matrixA.preMultiply(columnVectorX);
```

进行 $\boldsymbol{Ax}$ 运算时，通常希望结果是向量（而不是矩阵中的一个列向量）。若 $\boldsymbol{x}$ 是 RealVector 类型，则完成 $\boldsymbol{Ax}$ 运算更方便的方式如下：

```
/* Ax */
RealVector matrixVectorProduct = matrixA.operate(vectorX);
```

2.2.7　内积

内积（inner product，也称作点积或标量积）是一种计算两个相同维数向量乘积的方法，其

结果是标量值，在数学表达式中用两个向量之间的圆点来表示。

$$\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=1}^{n} a_i b_i$$

对于 RealVector 对象 vectorA 与 vectorB，点积计算如下：

```
double dotProduct = vectorA.dotProduct(vectorB);
```

若向量是矩阵形式，则可以用矩阵乘法来计算，这是因为 $\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}$，其中等号左侧是点积，右侧是矩阵相乘。

$$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \end{pmatrix} \begin{pmatrix} b_{1,1} \\ b_{2,1} \\ \vdots \\ b_{n,1} \end{pmatrix}$$

列向量 $\boldsymbol{a}$ 与 $\boldsymbol{b}$ 采用矩阵相乘，返回 1×1 矩阵。

```
/* matrixA与matrixB都是m × 1的列向量 */
RealMatrix innerProduct = matrixA.transpose().multiply(matrixB);

/* 计算结果保存在矩阵的唯一元素中 */
double dotProduct = innerProduct.getEntry(0,0);
```

与点积相比，此处采用矩阵相乘可能显得不太实用，但是它说明了向量运算与矩阵运算之间的一种重要关系。

2.2.8 外积

m 维向量 $\boldsymbol{a}$ 与 n 维向量 $\boldsymbol{b}$ 的**外积**（outer product）返回 $m \times n$ 的新矩阵。

$$\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}} = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \end{pmatrix} = \begin{pmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & \cdots & a_{1,1}b_{1,n} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & \cdots & a_{2,1}b_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}b_{1,1} & a_{m,1}b_{1,2} & \cdots & a_{m,1}b_{1,n} \end{pmatrix}$$

注意 $\boldsymbol{a}\boldsymbol{b}^{\mathrm{T}}$ 的结果是 $m \times n$ 矩阵，且不等于 $\boldsymbol{b}\boldsymbol{a}^{\mathrm{T}}$，后者的结果是 $n \times m$ 矩阵。RealMatrix.outerProduct() 方法保持了这种顺序，它返回具有适当大小的 RealMatrix 实例。

```
/* 向量a与向量b的外积 */
RealMatrix outerProduct = vectorA.outerProduct(vectorB);
```

若向量是矩阵形式，则可以改为用 RealMatrix.multiply() 计算外积。

```
/* matrixA与matrixB都是n × 1的列向量 */
RealMatrix outerProduct = matrixA.multiply(matrixB.transpose());
```

2.2.9 逐项积

逐项积也称作 Hadamard 积或者 Schur 积。逐项积是把向量的每个分量与另一向量的对应分量相乘，两个向量必须有相同的维数，所产生的结果向量也有着与这两个向量相同的维数。

$$\boldsymbol{a} \circ \boldsymbol{b} = (a_1b_1, a_2b_2, \cdots, a_nb_n)$$

`RealVector.ebeMultiply(RealVector)` 方法可以完成这个运算，其中 ebe 是**逐项**（element by element）的缩写。

```
/* 计算向量a与向量b的逐项积 */
RealVector vectorATimesVectorB = vectorA.ebeMultiply(vectorB);
```

采用 `RealVector.ebeDivision(RealVector)` 方法可以完成类似的逐项除运算。



不要将逐项积与矩阵积混淆（包括内积与外积）。在多数算法中需要矩阵积。然而，逐项积也会有派上用场的时候，比如需要把向量用另一权值向量进行缩放时。

目前，在 Apache Commons Math 矩阵乘法中，并没有实现 Hadamard 积。但是，用下面的简单方法可以很容易地实现它：

```
public class MatrixUtils {

    public static RealMatrix ebeMultiply(RealMatrix a, RealMatrix b) {
        int rowDimension = a.getRowDimension();
        int columnDimension = a.getColumnDimension();
        RealMatrix output = new Array2DRowRealMatrix(rowDimension,
            columnDimension);
        for (int i = 0; i < rowDimension; i++) {
            for (int j = 0; j < columnDimension; j++) {
                output.setEntry(i, j, a.getEntry(i, j) * b.getEntry(i, j));
            }
        }
        return output;
    }
}
```

可以像下面这样实现 Hadamard 积：

```
/* matrixA与matrixB的逐项积 */
RealMatrix hadamardProduct = MatrixUtils.ebeMultiply(matrixA, matrixB);
```

2.2.10 复合运算

我们会经常遇见涉及多个向量与矩阵的复合形式，例如 $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{Ax}$，它将产生单一的标量值。有时成块地计算是方便的，甚至可以不按顺序计算。在这种情况下，可以首先计算向量 $\boldsymbol{v} = \boldsymbol{Ax}$，然后计算点积（内积）$\boldsymbol{x} \cdot \boldsymbol{v}$。

```
double[] xData = {1, 2, 3};
double[][] aData = {{1, 3, 1}, {0, 2, 0}, {1, 5, 3}};
RealVector vectorX = new ArrayRealVector(xData);
RealMatrix matrixA = new Array2DRowRealMatrix(aData);
double d = vectorX.dotProduct(matrixA.operate(vectorX));
// d = 78
```

另一个方案是首先调用 `RealMatrix.premultiply()` 方法，即用向量左乘矩阵，然后计算两个向量的内积（点积）。

```
double d = matrixA.premultiply(vecotrX).dotProduct(vectorX);
// d = 78
```

若向量是列向量的矩阵形式，则可以仅采用矩阵运算方法，但注意结果也是矩阵。

```
RealMatrix matrixX = new Array2DRowRealMatrix(xData);
/* 结果是1 × 1的矩阵 */
RealMatrix matrixD = matrixX.transpose().multiply(matrixA).multiply(matrixX);
d = matrixD.getEntry(0, 0); // 78
```

2.2.11 仿射变换

一个常见的操作是对向量 $\boldsymbol{x}$ 进行变换，首先用 $p \times n$ 的线性映射矩阵 $\boldsymbol{A}$ 左乘维数为 n 的向量 $\boldsymbol{x}$，再加上维数为 p 的平移向量 $\boldsymbol{b}$，它们的关系如下：

$$f(\boldsymbol{x}) = \boldsymbol{Ax} + \boldsymbol{b}$$

这种变换称作**仿射变换**（affine transformation）。为方便起见，可以令 $\boldsymbol{z} = f(\boldsymbol{x})$，把向量 $\boldsymbol{x}$ 移动到另一侧，定义 $\boldsymbol{W} = \boldsymbol{A}^{\mathrm{T}}$，$\boldsymbol{W}$ 是 $n \times p$ 的矩阵，从而有下式：

$$\boldsymbol{z}^{\mathrm{T}} = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{W} + \boldsymbol{b}^{\mathrm{T}}$$

在学习与预测算法中，尤其可以看到大量的这种形式。重要的是要注意到，$\boldsymbol{x}$ 是一次观测得到的一个多维向量，而不是许多次观测得到的一维向量。展开后的表达式如下：

$$\begin{pmatrix} z_1 \; z_2 \cdots z_p \end{pmatrix} = \begin{pmatrix} x_1 \; x_2 \cdots x_n \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} \beta_1 \; \beta_2 \cdots \beta_p \end{pmatrix}$$

$m \times n$ 矩阵 $\boldsymbol{X}$ 的仿射变换也可以表示如下：

$$\boldsymbol{Z} = \boldsymbol{XW} + \boldsymbol{B}$$

$\boldsymbol{B}$ 是 $m \times p$ 矩阵。

$$\boldsymbol{Z} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,p} \\ \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{m,1} & \beta_{m,2} & \cdots & \beta_{m,p} \end{pmatrix}$$

大多数情况下，我们希望平移矩阵的每一行与向量 $\boldsymbol{b}$（维数为 p）相同，于是表达式如下：

$$\boldsymbol{Z} = \boldsymbol{XW} + \boldsymbol{hb}^{\mathrm{T}}$$

其中 $\boldsymbol{h}$ 是维数为 m 的列向量，每个分量为 1。注意这两个向量的外积生成 $m \times p$ 矩阵，展开之后，表达式如下：

$$\boldsymbol{Z} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \begin{pmatrix} \beta_1 & \beta_2 \cdots \beta_p \end{pmatrix}$$

该功能非常重要，因此在 `MatrixOperations` 类中进行了实现。

```
public class MatrixOperations {
...
    public static RealMatrix XWplusB(RealMatrix x, RealMatrix w, RealVector b) {
        RealVector h = new ArrayRealVector(x.getRowDimension(), 1.0);
        return x.multiply(w).add(h.outerProduct(b));
    }
...
}
```

2.2.12　映射函数

我们通常需要在向量 $\boldsymbol{z}$ 的内容上映射函数 φ，结果是与 $\boldsymbol{z}$ 维数相同的新向量 $\boldsymbol{y}$。

$$\boldsymbol{y} = \varphi(\boldsymbol{z})$$

Commons Math API 中包含 `RealVector.map(UnivariateFunction function)` 方法，它可以完成上述功能。Commons Math 中包含了大多数标准函数以及其他一些有用的函数，它们实现了 `UnivariateFunction` 接口，可以采用下面的方法调用：

```
// 在向量input上映射函数exp，产生新的向量output
RealVector output = input.map(new Exp());
```

对于那些 Commons Math 中没有包括的函数形式，可以直接创建自己的 UnivariateFunction 类。注意 map 方法不更改输入向量，若想对输入向量进行就地更改，则要使用下面的方法：

```
// 在向量input上映射函数exp，从而修改它的值
input.mapToSelf(new Exp());
```

在有些场合，需要将一元函数应用到矩阵的每个元素。为此，Apache Commons Math API 提供了一种简洁的方法，即使是稀疏矩阵，这种方法也可以有效地工作，这就是 RealMatrix.walkInOptimizedOrder(RealMatrixChangingVisitor visitor) 方法。请注意，还有其他选项，可以按照行或列的顺序遍历矩阵的每个元素，这对于某些运算是有用的（或者是必需的）。不过，若只想独立地更新矩阵的每个元素，则使用优化顺序是最有适应能力的算法，因为它适用于采用二维数组、块或稀疏存储的矩阵。首先创建扩展 RealMatrixChangingVisitor 接口的类（充当映射函数）并实现所需的方法。

```
public class PowerMappingFunction implements RealMatrixChangingVisitor {

    private double power;

    public PowerMappingFunction(double power) {
        this.power = power;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
        int startColumn, int endColumn) {
        // 在运算开始之前调用一次……此处不需要
    }

    @Override
    public double visit(int row, int column, double value) {
        return Math.pow(value, power);
    }

    @Override
    public double end() {
        // 遍历所有项之后，调用一次……此处不需要
        return 0.0;
    }
}
```

然后把所需函数映射到已有矩阵，就像下面这样，把该类的实例传递到 walkInOptimizedOrder() 方法。

```
/* 矩阵的每个元素x都就地更新为x^1.2 */
matrix.walkInOptimizedOrder(new PowerMappingFunction(1.2));
```

也可以利用 Apache Commons Math 内置的分析函数，它实现了 UnivariateFunction 接口，可以轻松地把任意函数映射到矩阵的每个元素。

```
public class UnivariateFunctionMapper implements RealMatrixChangingVisitor {

    UnivariateFunction univariateFunction;

    public UnivariateFunctionMapper(UnivariateFunction univariateFunction) {
        this.univariateFunction = univariateFunction;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
            int startColumn, int endColumn) {
        // 不匹配
    }

    @Override
    public double visit(int row, int column, double value) {
        return univariateFunction.value(value);
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

例如，想要扩展 2.2.11 节中仿射变换的静态方法时，就可以使用这个接口：

```
public class MatrixOperations {
...
    public static RealMatrix XWplusB(RealMatrix X, RealMatrix W, RealVector b,
            UnivariateFunction univariateFunction) {

        RealMatrix z = XWplusB(X, W, b);
        z.walkInOptimizedOrder(new UnivariateFunctionMapper(univariateFunction));
        return z;
    }
...
}
```

因此，若想把 S 型函数（logistic 函数）映射到仿射变换，则可以这样做：

```
// 根据输入矩阵x、权值矩阵w以及偏移向量b生成新矩阵之后，
// 把S型函数映射到新矩阵的所有元素
MatrixOperations.XWplusB(x, w, b, new Sigmoid());
```

有几件重要的事情需要说明。首先，这里也有**保护访问者**（preserving visitor），它遍历矩阵每个元素，但是不改变其内容。另外需要注意的是方法。唯一真正需要实现的是 visit() 方法，它应当返回每个输入值的新值。此处并不需要 start() 与 end() 方法（尤其在这种情况下）。在启动所有操作之前只需要调用一次 start() 方法。例如，假设在后续计算中需要矩阵行列式，就可以在 start() 方法中计算一次，把它作为类变量存储，然后在 visit() 操作中使用它。同样，在访问过所有元素后，调用一次 end() 方法。可以用这个方

法来记录运行指标、访问站点总数，乃至错误信号。在任何情况下，当所有事情做完后，该方法返回 `end()` 的值。不需要在 `end()` 方法中引入任何实际逻辑，但是至少可以返回有效的 `double` 值，例如 0.0，它不过是个占位符而已。注意 `RealMatrix.walkInOptimizedOrder(RealMatrixChangingVisitor visitor, int startRow, int endRow, int startColumn, int endColumn)` 方法，它只针对子阵进行操作，子阵边界由方法签名给出。当只需要就地更新矩阵的特定矩形块，而保持其余位置不变时，可以使用该方法。

2.3 矩阵分解

考虑到我们已经掌握的有关矩阵相乘的知识，很容易想到，任何矩阵都可以分解为几个其他矩阵的乘积。把一个矩阵分解为几个矩阵，可以确保对重要的矩阵性质进行有效且数值稳定的计算。例如，尽管矩阵求逆以及矩阵行列式有显式的代数公式，但在对其计算时，最好还是先把它进行分解，然后再求逆。行列式可以直接从 Cholesky 分解或者 LU 分解中得到。此处所有矩阵分解都可以用于求解线性方程组，因而可以求矩阵逆。表 2-1 列出了 Apache Commons Math 中实现的各种矩阵分解的性质。

表2-1：矩阵分解的性质

分解算法	矩阵类型	求解方法	求逆	行列式
Cholesky	对称正定	精确	✓	✓
特征	方阵	精确	✓	✓
LU	方阵	精确	✓	✓
QR	任意	最小二乘法	✓	
SVD	任意	最小二乘法	✓	

2.3.1 Cholesky分解

矩阵 $\boldsymbol{A}$ 的 Cholesky 分解是指把矩阵 $\boldsymbol{A}$ 分解为 $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$，其中 $\boldsymbol{L}$ 是下三角矩阵，其上三角（对角线之上）为零：

$$\boldsymbol{A} = \begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{1,2} & \cdots & l_{1,n} \\ 0 & l_{2,2} & \cdots & l_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{n,n} \end{pmatrix}$$

```
CholeskyDecomposition cd = new CholeskyDecomposition(matrix);
RealMatrix l = cd.getL();
```

Cholesky 分解只对对称正定矩阵有效，其主要用途是计算服从多维正态分布的随机变量。

2.3.2 LU分解

下-上（LU）分解［lower-upper（LU）decomposition］把矩阵 $\boldsymbol{A}$ 分解为下三角矩阵 $\boldsymbol{L}$ 与上三角矩阵 $\boldsymbol{U}$，即 $\boldsymbol{A}=\boldsymbol{LU}$：

$$\boldsymbol{A}=\begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix}\begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix}$$

```
LUDecomposition lud = new LUDecomposition(matrix);
RealMatrix u = lud.getU();
RealMatrix l = lud.getL();
```

LU 分解用于求解线性方程组，其中未知数个数与方程数相同。

2.3.3 QR分解

QR 分解把矩阵 $\boldsymbol{A}$ 分解成由单位列向量组成的正交矩阵 $\boldsymbol{Q}$ 以及上三角矩阵 $\boldsymbol{R}$：

$$\boldsymbol{A}=\begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,m} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ q_{m,1} & q_{m,2} & \cdots & q_{m,m} \end{pmatrix}\begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{m,n} \end{pmatrix}$$

```
QRDecomposition qrd = new QRDecomposition(matrix);
RealMatrix q = lud.getQ();
RealMatrix r = lud.getR();
```

QR 分解（以及类似分解）的主要应用之一是计算特征分解，这是因为矩阵 $\boldsymbol{Q}$ 的每个列是正交的。QR 分解也用于求解超定线性方程组。对数据点的数目（行数）大于维数（列数）的数据集而言，通常就属于这种情况。使用 QR 分解求解（相对于 SVD）的优点之一是容易获得解参数误差，解参数可以直接从矩阵 $\boldsymbol{R}$ 中计算出。

2.3.4 奇异值分解

奇异值分解（SVD）是指把 $m \times n$ 矩阵 $\boldsymbol{A}$ 分解为 $\boldsymbol{A}=\boldsymbol{USV}^{\mathrm{T}}$。其中，$\boldsymbol{U}$ 是 $m \times m$ 酉矩阵；$\boldsymbol{S}$ 是 $m \times n$ 对角矩阵，其元素是非负实数；$\boldsymbol{V}$ 是 $n \times n$ 酉矩阵。作为酉矩阵，$\boldsymbol{U}$ 与 $\boldsymbol{V}$ 都具有 $\boldsymbol{UU}^{\mathrm{T}}=\boldsymbol{I}$ 的性质，其中 $\boldsymbol{I}$ 是单位矩阵。

$$\boldsymbol{A}=\begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,m} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m,1} & u_{m,2} & \cdots & u_{m,m} \end{pmatrix}\begin{pmatrix} s_{1,1} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & s_{2,2} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \\ 0 & 0 & \cdots & s_{n,n} & \cdots & 0_{m,n} \end{pmatrix}\begin{pmatrix} v_{1,1} & v_{2,1} & \cdots & v_{n,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,n} & v_{2,n} & \cdots & v_{n,n} \end{pmatrix}$$

大多数情况下，$m \geqslant n$，即矩阵中的行数大于或等于列数。这样就没必要计算完整的 SVD，而是可以实现更有效的计算，称作**瘦**（thin）SVD。其中，$\boldsymbol{U}$ 为 $m \times n$ 矩阵，$\boldsymbol{S}$ 为 $n \times n$ 矩阵，$\boldsymbol{V}$ 为 $n \times n$ 矩阵。在实际中，也可能是 $m \leqslant n$，于是此时可以只采用两个维数中的较小值，即 $p = \min(m, n)$。Apache Commons Math 的实现就用了这个方法。

```
/* 矩阵matrix是m × n的，且p = min(m, n) */
SingularValueDecomposition svd = new SingularValueDecomposition(matrix);
RealMatrix u = svd.getU(); // m × p
RealMatrix s = svd.getS(); // p × p
RealMatrix v = svd.getV(); // p × n
/* 从S的对角线以降序的方式获取值 */
double[] singularValues = svd.getSingularValues();
/* 也可以获得输入矩阵的协方差矩阵 */
double minSingularValue = 0;// 零或负值意味着使用所有的奇异值
RealMatrix cov = svd.getCovariance(minSingularValue);
```

奇异值分解有几个有用的性质。类似于特征分解，它用于对矩阵 $\boldsymbol{A}$ 降维，只保留那些最有用的维。此外，作为一种线性解法，SVD 可以用于任何形状的矩阵。尤其是在求解欠定矩阵时，即维数（列数）远大于数据点数（行数）时，SVD 是稳定的。

2.3.5 特征分解

特征（eigen）分解的目标是把矩阵 $\boldsymbol{A}$ 重新组织为独立且正交的列向量的集合，称作**特征向量**（eigenvector）。每个特征向量都有相应的特征值，可用于对特征向量按从最重要（最大特征值）到最不重要（最小特征值）的顺序来排序。然后，可以只选择最重要的特征向量作为矩阵 $\boldsymbol{A}$ 的代表。一个根本问题是，有没有什么方法可以只用较少维数就能描述矩阵 $\boldsymbol{A}$ 的全部（或大部分）？

对于矩阵 $\boldsymbol{A}$，存在由向量 $\boldsymbol{x}$ 以及常数 λ 组成的解，使得 $\boldsymbol{Ax} = \lambda\boldsymbol{x}$。可以存在多个解（即 $\boldsymbol{x}, \lambda$ 对）。λ 的所有可能值的集合称为特征值，所有对应的向量称为特征向量。对称实矩阵 $\boldsymbol{A}$ 的特征分解表示为 $\boldsymbol{A} = \boldsymbol{VDV}^{\mathrm{T}}$，结果可表示为 $m \times m$ 对角矩阵 $\boldsymbol{D}$（其中特征值在对角线上）和 $m \times m$ 矩阵 $\boldsymbol{V}$（其列向量即特征向量）。

$$\boldsymbol{A} = \begin{pmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,m} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \cdots & v_{m,m} \end{pmatrix} \begin{pmatrix} d_{1,1} & 0 & \cdots & 0 \\ 0 & d_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{m,m} \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{2,1} & \cdots & v_{m,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,m} & v_{2,m} & \cdots & v_{m,m} \end{pmatrix}$$

有几种方法进行特征分解。从实用角度讲，通常只需要 Apache Commons Math 在 `org.apache.commons.math3.linear.EigenDecomposition` 类中实现的最简单形式即可。特征值与特征向量采用特征值降序的方式排列。换句话说，第一个特征向量（矩阵 $\boldsymbol{V}$ 的第 0 列）是最重要的特征向量。

```
double[][] data = {{1.0, 2.2, 3.3}, {2.2, 6.2, 6.3}, {3.3, 6.3, 5.1}};
RealMatrix matrix = new Array2DRowRealMatrix(data);

/* 计算特征值矩阵D以及特征向量矩阵V */
EigenDecomposition eig = new EigenDecomposition(matrix);

/* 特征值的实部（或虚部）可以用double型的数组获得 */
double[] eigenValues = eig.getRealEigenvalues();

/* 也可以直接从矩阵D中访问单个的特征值 */
double firstEigenValue = eig.getD().getEntry(0, 0);

/* 可以用这种方式访问第一个特征向量 */
RealVector firstEigenVector = eig.getEigenvector(0);

/* 记住特征向量正是矩阵V的列 */
RealVector firstEigenVector = eig.getV.getColumn(0);
```

2.3.6 行列式

行列式（determinant）是标量值，它由矩阵 $\boldsymbol{A}$ 计算得出，通常在多维正态分布中需要计算行列式。矩阵 $\boldsymbol{A}$ 的行列式记作 $|\boldsymbol{A}|$。Cholesky 分解、特征分解以及 LU 分解的类提供了得到行列式的方法。

```
/* 从Cholesky分解中计算行列式 */
double determinant = new CholeskyDecomposition(matrix).getDeterminant();

/* 从特征分解中计算行列式 */
double determinant = new EigenDecomposition(matrix).getDeterminant();

/* 从LU分解中计算行列式 */
double determinant = new LUDecomposition(matrix).getDeterminant();
```

2.3.7 矩阵逆

矩阵逆（inverse）的概念类似于实数 $\Re$ 的倒数，其中 $\Re(1/\Re) = 1$。注意该式也可以写作 $\Re\Re^{-1} = 1$。同样，矩阵 $\boldsymbol{A}$ 的逆记作 $\boldsymbol{A}^{-1}$，矩阵与矩阵逆存在关系 $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}$，其中 $\boldsymbol{I}$ 是单位矩阵。虽然有直接计算矩阵逆的公式，但是对大矩阵而言，这些公式缓慢而复杂，数值计算也不稳定。Apache Commons Math 中的每种分解方法都实现了 `DecompositionSolver` 接口，该接口在求解线性方程组时需要矩阵逆。于是，矩阵逆可以通过 `DecompositionSolver` 类的访问方法获得。若矩阵类型与所使用的方法相容，则任何分解方法都可求出矩阵逆。

```
/* 用Cholesky分解、LU分解、特征分解、QR分解或SVD分解求方阵的逆 */
RealMatrix matrixInverse = new LUDecomposition(matrix).getSolver().getInverse();
```

矩阵逆也可以从奇异值分解中计算得出：

```
/* 用SVD分解求方阵或矩形矩阵的逆 */
RealMatrix matrixInverse =
new SingularValueDecomposition(matrix).getSolver().getInverse();
```

或者也可以使用 QR 分解：

```
/* 适用于矩形矩阵，但是用于非奇异矩阵时会出错 */
RealMatrix matrixInverse = new QRDecomposition(matrix).getSolver().getInverse();
```

每当通过除法把矩阵从等式一侧移动到另一侧时，会用到矩阵逆。另一个常见应用是计算 Mahalanobis 距离，以及通过扩展计算多维正态分布。

2.4 求解线性方程组

本章一开始描述了系统 $\boldsymbol{XW} = \boldsymbol{Y}$，并把它作为线性代数的基本概念。通常，也需要引入不依赖于 x 的 β 项——截距项或偏移量，如下式：

$$y_{1,1} = \beta + x_{1,1}\omega_{1,1} + x_{1,2}\omega_{2,1} + \cdots + x_{1,n}\omega_{n,1}$$

有两种方法引入截距项。第一种是为矩阵 $\boldsymbol{X}$ 添加一个全 1 的列，以及为矩阵 $\boldsymbol{W}$ 添加一行未知数。在这种情况下，选择哪个行列对并不重要，只要 $j = i$ 就可以了。此处选择 $\boldsymbol{X}$ 的最后一列以及 $\boldsymbol{W}$ 的最后一行。

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} & 1 \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n+1,1} & \omega_{n+1,2} & \cdots & \omega_{n+1,p} \end{pmatrix} \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{1,2} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,p} \end{pmatrix}$$

注意，在这种情况下，$\boldsymbol{W}$ 的列是独立的。因此，只是为了可以方便地在一段代码中完成运算，我们寻找了 p 个独立的线性模型。

```
/* 数据 */
double[][] xData = {{0, 0.5, 0.2}, {1, 1.2, .9}, {2, 2.5, 1.9}, {3, 3.6, 4.2}};
double[][] yData = {{-1, -0.5}, {0.2, 1}, {0.9, 1.2}, {2.1, 1.5}};

/* 创建矩阵X，偏移量作为最后一列 */
double[] ones = {1.0, 1.0, 1.0, 1.0};
int xRows = 4;
int xCols = 3;
RealMatrix x = new Array2DRowRealMatrix(xRows, xCols + 1);
x.setSubMatrix(xData, 0, 0);
x.setColumn(3, ones); // 第4列的索引值为3

/* 创建矩阵Y */
RealMatrix y = new Array2DRowRealMatrix(yData);
```

```
/* 求解矩阵W的值 */
SingularValueDecomposition svd = new SingularValueDecomposition(x);
RealMatrix solution = svd.getSolver().solve(y);
System.out.println(solution);
// {{1.7,3.1},{-0.9523809524,-2.0476190476},
// {0.2380952381,-0.2380952381},{-0.5714285714,0.5714285714}}
```

给定参数值，方程组的解如下：

$$y_1 = 1.7\,x_1 - 0.95\,x_2 + 0.24\,x_3 - 0.57$$

$$y_2 = 3.1\,x_1 - 2.05\,x_2 - 0.24\,x_3 + 0.57$$

如果意识到前面的代数表达式等价于本章早先讲过的矩阵仿射变换，那就可以得到引入截距的第二种方法：

$$\boldsymbol{Y} = \boldsymbol{XW} + \boldsymbol{hb}^{\mathrm{T}}$$

这种形式的线性系统，优点是不需要调整任何矩阵的大小。前面的示例代码中只调整了一次矩阵，这并不会造成太大负担。然而，在第 5 章处理多层线性模型（深度网络）时，调整矩阵将是困难且低效的。在那种情况下，更方便的选择是用代数项表示线性模型，其中 $\boldsymbol{W}$ 与 $\boldsymbol{b}$ 是完全独立的。

第3章 统计学

把统计学的基本原理应用到数据科学，可以深刻地了解数据。统计学是个强大的工具，使用得当的话，就可以使人们很有把握地做出决策。然而，统计学很容易误用。此处的示例是 Anscombe 的 4 组数据（见图 3-1），图中显示了 4 个截然不同的数据集，它们却有着几乎相同的统计值。在许多情况下，数据的简单图示可以立刻使人们意识到数据意味着什么。从 Anscombe 的 4 组数据中可以立刻得到这些特征：左上角的图中，x 与 y 是线性的，但有异常值；右上角的图中，可以看到 x 与 y 形成一种有顶点的非线性关系；左下角的图中，除了一个异常值之外，x 与 y 是准确的线性关系；右下角的图中，在统计上，y 分布在 $x = 8$ 上，但是有个异常值在 $x = 19$ 处。尽管每幅图看上去都是如此不同，但对每个数据集进行标准统计计算，其结果却相同。显然，在现实中，眼睛是最精妙的数据处理工具！然而，不能总是用这种方式将数据可视化。许多时候数据是多维的，也许 x 是多维的，也许 y 也是多维的。尽管可以对 x 与 y 的每个维度绘制图形，获得数据集的某些特征，却会丢掉 x 中各变量之间的所有依赖关系。



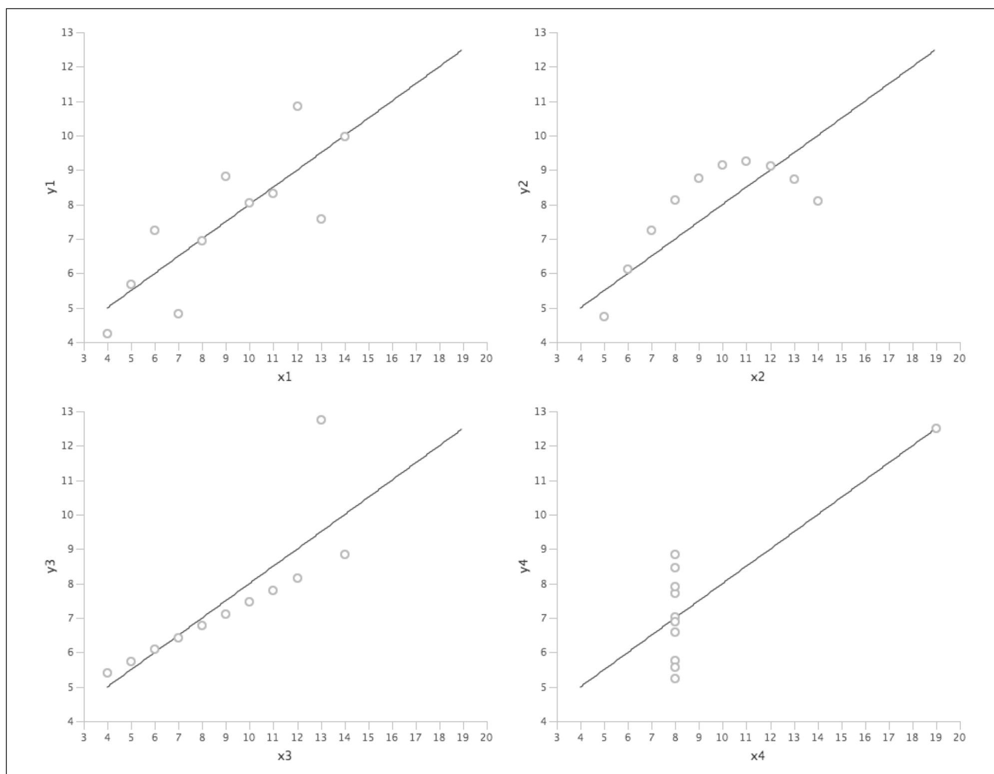


图 3-1：Anscombe 的 4 组数据

3.1　数据的概率起源

本书的开端把**数据点**（data point）定义为一个事件的记录，该事件发生在特定时间、特定地点。可以用狄拉克 δ 函数 $\delta(\boldsymbol{x})$ 表示数据点，它在 $\boldsymbol{x}=0$ 时值为 ∞，在其他点时值为 0。该函数可以进一步推广为 $\delta(x-x_i)$，这意味着狄拉克 δ 函数在 $x=x_i$ 时值为 ∞，在其他点时值为 0。这里有个问题：是什么促使了数据点的产生？

3.1.1　概率密度

有时数据来自熟知的生成源，它可以用函数 $f(x)$ 描述。$f(x)$ 通常可以通过一些参数 θ 进行调整，记作 $f(x;\theta)$。存在多种 $f(x)$，它们中的大多数来自对自然世界行为的观察。接下来的几节将探究一些更常见的形式，包括连续分布以及离散随机数分布。

可以把每个位置的所有概率进行累加，成为变量 x 的函数。

$$f(x)=\sum_{i=1}^{n}p_i\delta(x-x_i)$$

或者对于离散整数变量 k，有下式：

$$f(x)=f(k)\delta(x-k)$$

注意 $f(x)$ 可以大于 1。概率密度并不是概率，而是局部密度。为了确定概率，必须在 x 的任意范围内对概率密度进行积分。通常用累积分布函数完成此项任务。

3.1.2　累积概率

概率分布函数（PDF，probability distribution function）需要进行恰当的归一化，使得对空间整体积分时，事件在整个空间发生的概率是 100%。

$$F=\int_{-\infty}^{\infty}f(x)\mathrm{d}x=1$$

然而，若事件尚未发生，则也可以计算出该事件将在 x 点发生的累积概率。

$$F(x)=\int_{-\infty}^{x}f(x')\mathrm{d}x'$$

注意累积分布函数是单调的（总是随着 x 的增长而增长），并且它（几乎）总是 S 型函数形状（倾斜的 S）。给定事件尚未发生，它将在 x 点发生的概率是多少？对于大的 x 值，$P=1$。强加这个条件（大的 x 值，$P=1$），就可以确保在某个已定义的区间中事件肯定发生。

邮
电

3.1.3　统计矩

对已知概率分布 $f(x)$ 求积分，可以得出累积分布函数（或者对整个空间求积分得到 1），在求积分时引入 x 的幂次，称作统计矩。对于已知统计分布，统计矩是围绕着中心点 c 的 k 阶期望，可以用下式计算：

$$\mu_k=\int_{-\infty}^{\infty}(x-c)^k f(x)\mathrm{d}x$$

对于 $k=1$ 的一阶矩，在 $c=0$ 时的特别量，就是 x 的期望或者均值：

$$\mu=\int_{-\infty}^{\infty}xf(x)\mathrm{d}x$$

与均值相关的 $k>1$ 的高阶矩，称作关于均值的**中心矩**（central moment），它们与描述性的统计值有关，可以表示如下：

$$\mu_{k>1}=\int_{-\infty}^{\infty}(x-\mu)^k f(x)\mathrm{d}x$$

均值的二阶、三阶以及四阶中心矩具有实用的统计含义。我们定义方差 σ^2 为二阶矩：

$$\sigma^2 = \mu_2$$

其平方根是标准差 σ，是数据距离均值多远的一种度量。**偏度**（skewness）γ 是表示分布非对称性的一种度量，与均值的三阶矩有关：

$$\gamma = \frac{\mu_3}{\sigma^3}$$

峰度（kurtosis）是一种表示分布尾部有多宽的度量，与均值的四阶中心矩有关：

$$\kappa = \frac{\mu_4}{\sigma^4}$$

下一节会探讨正态分布，这是一种最常用且普遍的概率分布。正态分布的峰度 $\kappa = 3$。因为我们经常把事物与正态分布相比较，所以术语**超值峰度**（excess kurtosis）定义如下：

$$\kappa = \frac{\mu_4}{\sigma^4} - 3$$

我们现在定义了关于正态分布的**峰度**（尾部的宽度）。注意，许多时候提及峰度时，实际上指的是超值峰度，两个术语可以互换。

定义高阶矩是可能的，并且在数据科学之外的领域，它有不同的用法与应用。本书只讨论到四阶矩。

3.1.4 熵

在统计学中，**熵**（entropy）是表示分布中信息不可预测性的一种度量。对于连续分布，熵如下式所示：

$$\mathcal{H}(p) = \int_{-\infty}^{\infty} p(x)\log_b(p(x))\mathrm{d}x$$

对于离散分布，熵如下式所示：

$$\mathcal{H}(p) = -\sum_i p(x_i)\log_b(p(x_i))$$

在熵的示例图中（见图 3-2）可以看到，当概率是 0 或 1 时，熵最低。在 $p = 0.5$ 时，熵最高。p 等于 0 与 1 时熵相同。

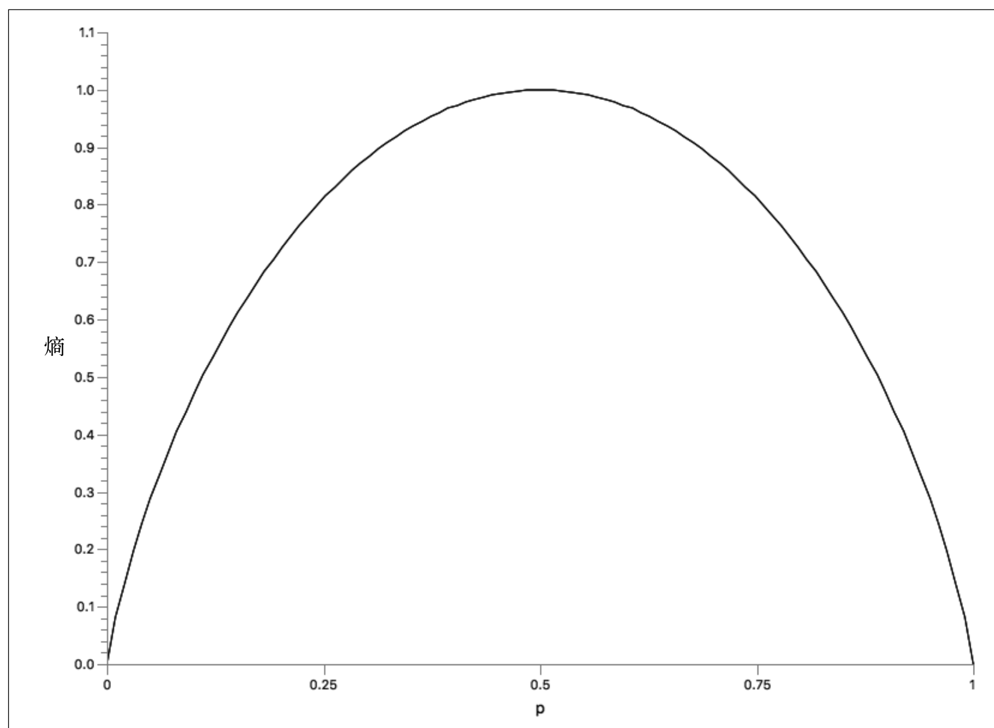


图 3-2：伯努利分布的熵

也可以采用交叉熵（cross entropy）来讨论两种分布之间的熵，其中 $p(x)$ 是真实的分布，而 $q(x)$ 是用于测试的分布：

$$\mathscr{H}(p,q)=\int_{-\infty}^{\infty}p(x)\log_b(q(x))\mathrm{d}x$$

对于离散的情形，有下式：

$$\mathscr{H}(p,q)=-\sum_i p(x_i)\log_b(q(x_i))$$

3.1.5　连续分布

一些众所周知的分布被很好地表征并得到广泛应用。许多分布是从真实世界自然现象观测中得到的。无论变量是用实数还是用整数描述，以表明相应的连续分布与离散分布，计算累积概率、统计矩以及统计度量的基本原则都是相同的。

1. 均匀分布

均匀分布（uniform distribution）在其定义的区间 $x \in [a, b]$ 上具有恒定不变的概率密度，

而在其他区间的概率密度为 0。事实上，这正是你所熟悉的生成 `[0, 1]` 区间上随机实数的正式描述，例如 `java.util.Random.nextDouble()`。默认的构造方法设置下限 $a = 0.0$，上限 $b = 1.0$。均匀分布概率密度的图形是**高帽**（top hat）或**箱**（box）的形状，如图 3-3 所示。

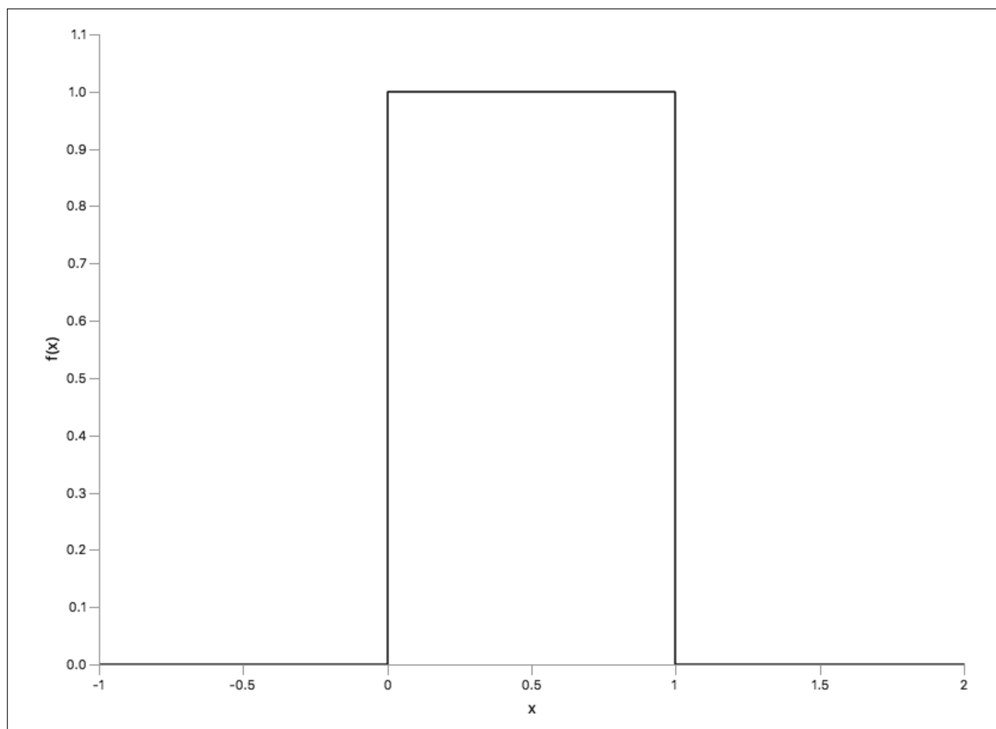


图 3-3：均匀分布的 PDF，参数 $a = 0$，$b = 1$

其 PDF 数学表达式如下：

$$f(x)=\begin{cases}\dfrac{1}{b-a}, & x\in[a,b]\\ 0, & \text{其他情况}\end{cases}$$

累积分布函数（CDF）如图 3-4 所示，其数学表达式如下：

$$F(x)=\begin{cases}0, & x<a\\ \dfrac{x-a}{b-a}, & x\in[a,b)\\ 1, & x\geqslant b\end{cases}$$

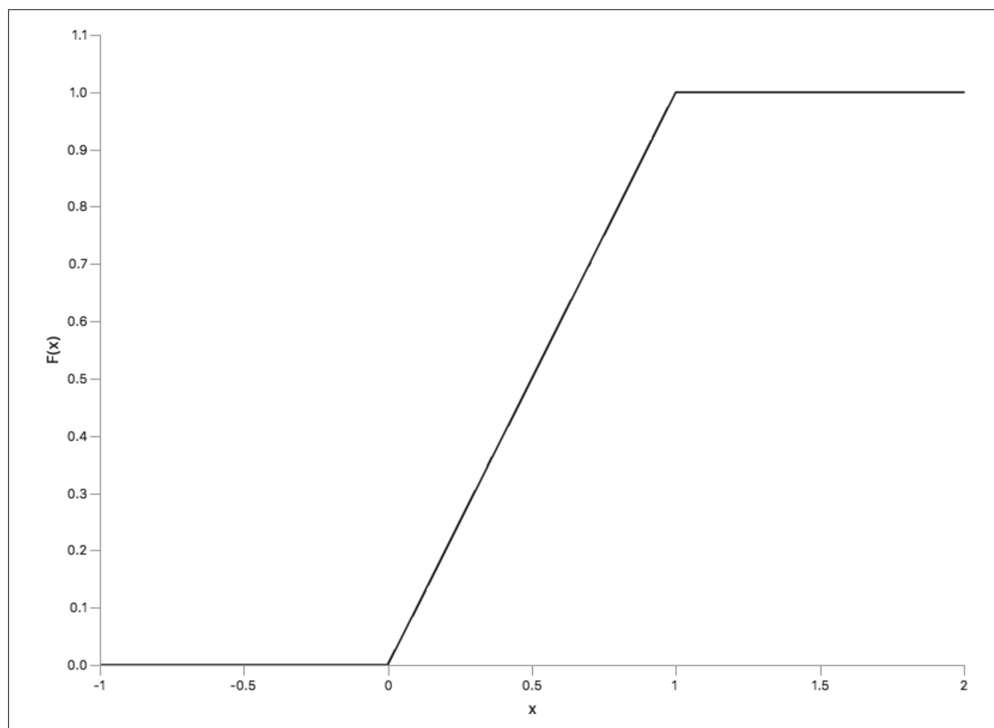


图 3-4：均匀分布的 CDF，参数 $a=0$，$b=1$

对于均匀分布，均值与方差不能直接给出，但是可以用下限与上限计算得出，如下式所示：

$$\mu=\frac{1}{2}(a+b)$$

$$\sigma^2=\frac{1}{12}(b-a)^2$$

要用 Java 调用均匀分布，可以使用 `UniformDistribution(a, b)` 类，其中下限与上限在构造方法中给出。若构造方法中不给出参数，则将调用标准均匀分布，其中 $a=0.0$，$b=1.0$。

```
UniformRealDistribution dist = new UniformRealDistribution();
double lowerBound = dist.getSupportLowerBound(); // 0.0
double upperBound = dist.getSupportUpperBound(); // 1.0
double mean = dist.getNumericalMean();
double variance = dist.getNumericalVariance();
double standardDeviation = Math.sqrt(variance);
double probability = dist.density(0.5));
double cumulativeProbability = dist.cumulativeProbability(0.5);
double sample = dist.sample(); // 例如0.023
double[] samples = dist.sample(3); // 例如{0.145,0.878,0.431}
```

注意，可以用 $a = \mu - \delta$ 与 $b = \mu + \delta$ 重新设置均匀分布的参数，其中 μ 是中心点（均值），δ 是从中心点到下限或上限的距离。于是，方差是 $\sigma^2 = \frac{\delta^2}{3}$，标准差是 $\sigma = \frac{\delta}{\sqrt{3}}$。那么，PDF 如下式所示：

$$f(x) = \begin{cases} \frac{1}{2\delta}, & x \in [\mu - \delta, \mu + \delta] \\ 0, & \text{其他情况} \end{cases}$$

CDF 如下式所示：

$$F(x) = \begin{cases} 0, & x < \mu - \delta \\ \frac{1}{2}(1 + \frac{x - \mu}{\delta}), & x \in [\mu - \delta, \mu + \delta) \\ 1, & x \geqslant \mu + \delta \end{cases}$$

基于中心点表示均匀分布，要计算 $a = \mu - \delta$ 与 $b = \mu + \delta$，并把它们写到构造方法中。

```
/* 初始化基于中心点的均匀分布，其中均值 = 10，半宽度 = 2 */
double mean = 10.0
double hw = 2.0;
double a = mean - hw;
double b = mean + hw;
UniformRealDistribution dist = new UniformRealDistribution(a, b);
```

此时，所有方法将返回正确结果，不需要进一步更改。在试图对分布进行比较时，这种围绕着均值重新设置参数的方法会派上用场。基于中心点的均匀分布是从正态分布（或其他关于峰值对称的分布）自然推广而来的。

2. 正态分布

在形形色色的应用场合中，正态分布是最有用且应用最广泛的分布。正态分布也称作**高斯分布**（Gaussian distribution）或**钟形曲线**（bell curve），这种分布是关于中心顶点对称的，其宽度可变。在多数场合中，说某事物的大小是围绕着平均值加上或减去特定量，指的正是正态分布。例如，对于班级考试成绩，一种解释是少数人考得确实好，少数人考得确实糟糕，但是大多数人考得一般或正好位于中间。在正态分布中，分布中心是最大的顶点，也是分布均值 μ，宽度用参数 σ 表示，同时它也是所有值的标准差。正态分布在 $x \in [-\infty, \infty]$ 都有定义，如图 3-5 所示。

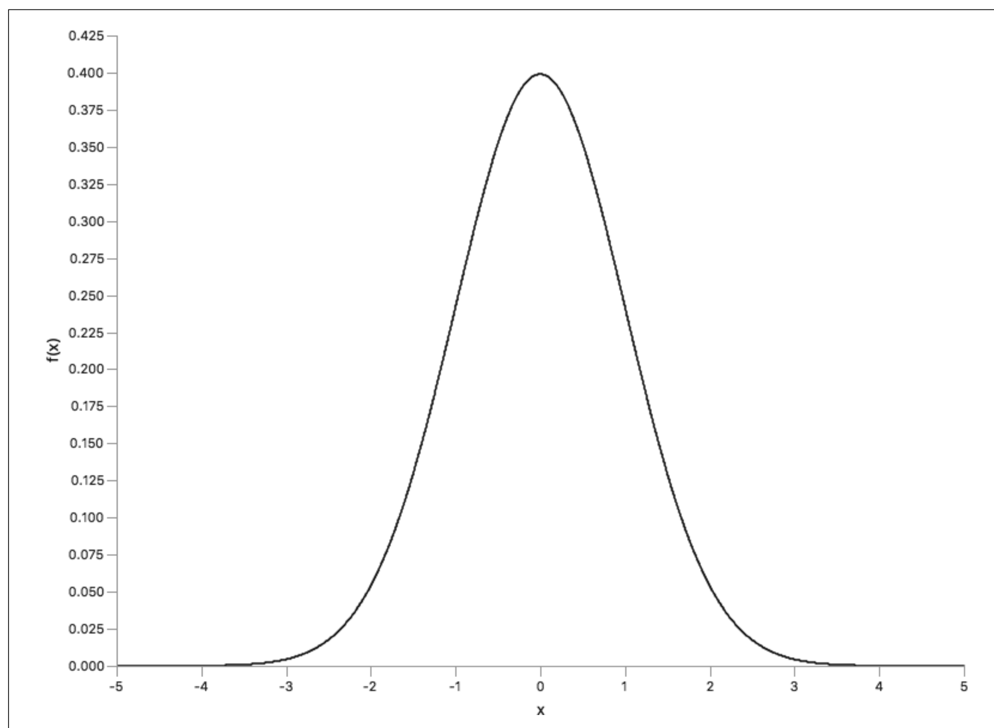


图 3-5：正态分布的 PDF，参数 $\mu = 0$，$\sigma = 1$

概率密度数学表达式如下：

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

累积分布函数形状如图 3-6 所示，它可以用误差函数表示：

$$F(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

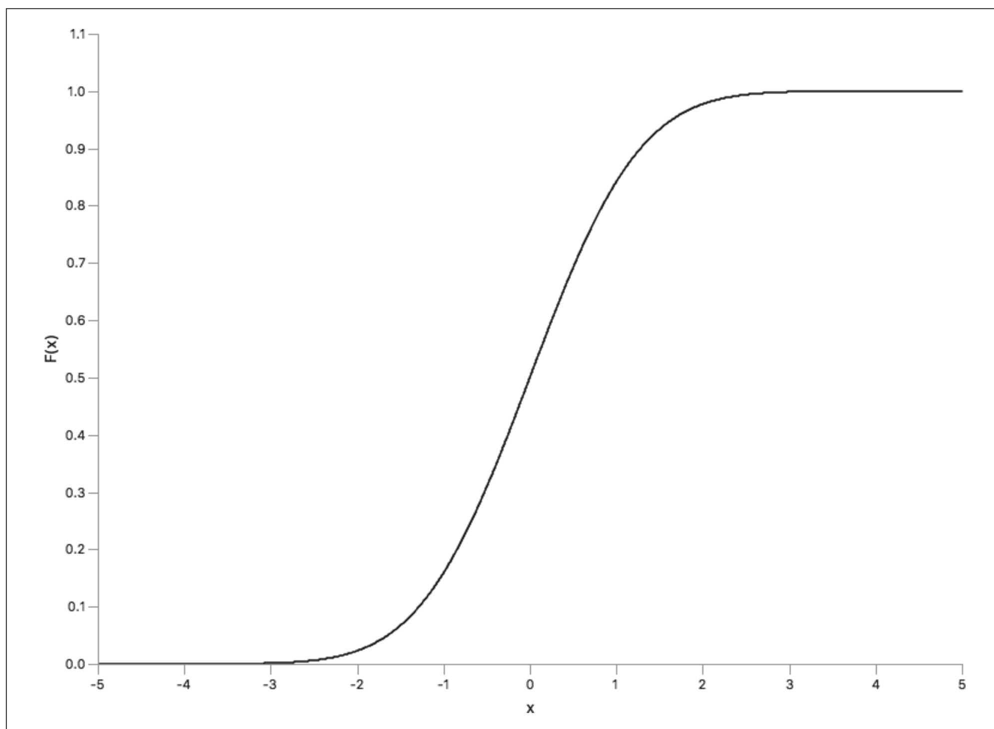


图 3-6：正态分布的 CDF，参数 $\mu = 0$，$\sigma = 1$

在 Java 中调用时，默认构造方法创建的就是标准正态分布，其中 $\mu = 0.0$，$\sigma = 1.0$。否则，可以把参数 μ 和 σ 传入构造方法中。

```
/* 采用默认的μ = 0以及σ = 1进行初始化 */
NormalDistribution dist = new NormalDistribution();
double mu = dist.getMean(); // 0.0
double sigma = dist.getStandardDeviation(); // 1.0
double mean = dist.getNumericalMean(); // 0.0
double variance = dist.getNumericalVariance(); // 1.0
double lowerBound = dist.getSupportLowerBound(); // 负无穷大
double upperBound = dist.getSupportUpperBound(); // 正无穷大
/* x = 0.0的概率密度 */
double probability = dist.density(0.0);
/* 计算x = 0.0的累积分布函数值 */
double cumulativeProbability = dist.cumulativeProbability(0.0);
double sample = dist.sample(); // 1.0120001
double samples[] = dist.sample(3); // {.0102, -0.009, 0.011}
```

3. 多维正态分布

正态分布可以推广到更高的维度，成为**多维正态**（multivariate normal，又名 multinormal）分布。变量 $\boldsymbol{x}$ 以及均值 $\boldsymbol{\mu}$ 是向量，而协方差矩阵 $\boldsymbol{\Sigma}$ 包含对角线上的方差以及其他 i, j 对对应的协方差。大体上，多维正态分布的形状是挤压的球形，且关于均值对称。当协方差为

0 且方差相等时，对于单位正态分布，该分布是完美的圆（或球）。服从该分布的随机点示例如图 3-7 所示。

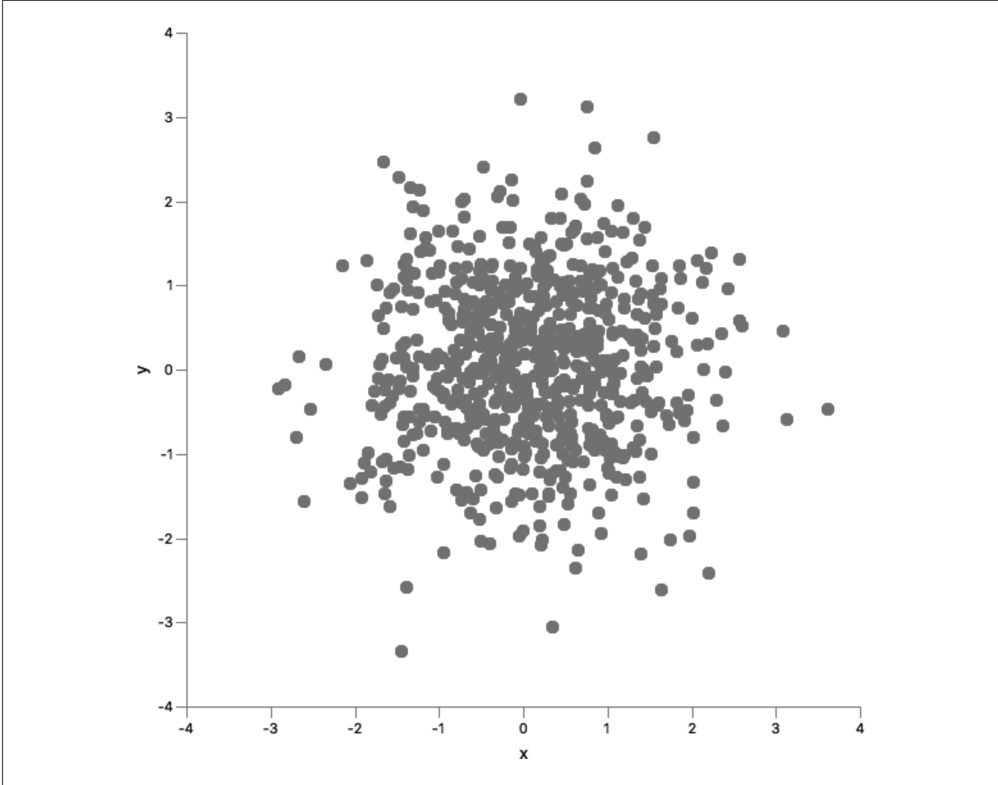


图 3-7：用二维正态分布产生的随机点

p 维正态分布的概率分布函数表达式为：

$$f(\boldsymbol{x})=\frac{1}{(2\pi)^{p/2}|\boldsymbol{\Sigma}|^{1/2}}\exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right)$$

注意，若协方差矩阵的行列式等于 0，即 $|\boldsymbol{\Sigma}|=0$ 则 $f(\boldsymbol{x})$ 将变为无穷大。同样要注意，当 $|\boldsymbol{\Sigma}|=0$ 时，不可能计算出所需的协方差矩阵逆 $\boldsymbol{\Sigma}^{-1}$ 。在这种情况下，称该矩阵是奇异的。若发生这样的情况，Apache Commons Math 将抛出下面的异常：

```
org.apache.commons.math3.linear.SingularMatrixException: matrix is singular
```

协方差矩阵是奇异的，这是什么原因造成的呢？这是一种共线性的征候，即基础数据的两个（或更多）变量相同，或者可以互相线性组合。换句话说，若有三个维度的数据，且协方差矩阵是奇异的，则可能意味着数据分布可以用两个维度甚至一个维度更好地描述。

CDF 没有对应的解析表达式，但可以通过数值积分得到。不过，Apache Commons Math 只支持一元数值积分。

虽然多维正态分布的均值以及协方差是 `double` 型的数组，但依旧可以通过 `MultivariateNormalDistribution` 类的 `getMeans()` 方法获得多维正态分布的均值，以及通过 `RealMatrix` 类的 `getData()` 方法获得协方差。

```
double[] means = {0.0, 0.0, 0.0};
double[][] covariances = {{1.0, 0.0, 0.0},{0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}};
MultivariateNormalDistribution dist =
    new MultivariateNormalDistribution(means, covariances);

/* 在点x = {0.0, 0.0, 0.0}的概率密度 */
double probability = dist.density(x); // 0.1
double[] mn = dist.getMeans();
double[] sd = dist.getStandardDeviations();
/* 返回RealMatrix，但是可以转换为double型数组 */
double[][] covar = dist.getCovariances().getData();
double[] sample = dist.sample();
double[][] samples = dist.sample(3);
```

注意有一种特殊情况，当变量完全独立时，协方差矩阵是对角矩阵。$\boldsymbol{\Sigma}$ 的行列式只是对角线上元素 $\sigma_{i,i}$ 的乘积。对角矩阵的逆仍然是对角矩阵，每个元素为 $1/\sigma_{i,i}$。于是，PDF 精简为一维正态分布 PDF 的乘积：

$$f(\boldsymbol{x}) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}\right)$$

在单位正态分布的情形中，多维单位正态分布的均值向量的分量为 0，协方差矩阵等于单位矩阵，即对角线上全为 1 的矩阵。

4. 对数正态分布

对数正态分布（log normal distribution）与正态分布的关系是，变量 x 的对数，即 $\ln(x)$ 的分布是正态分布。若在正态分布中用 $\ln(x)$ 替代 x，就可以得到对数正态分布，但有一些细微的区别。因为对数只对正数 x 有定义，所以这个分布的区间是 $x \in (0, \infty]$，即 $x > 0$。这个分布关于顶点是非对称的，在 x 较小时达到顶点，在 x 变大时有向无穷延展的长尾，如图 3-8 所示。

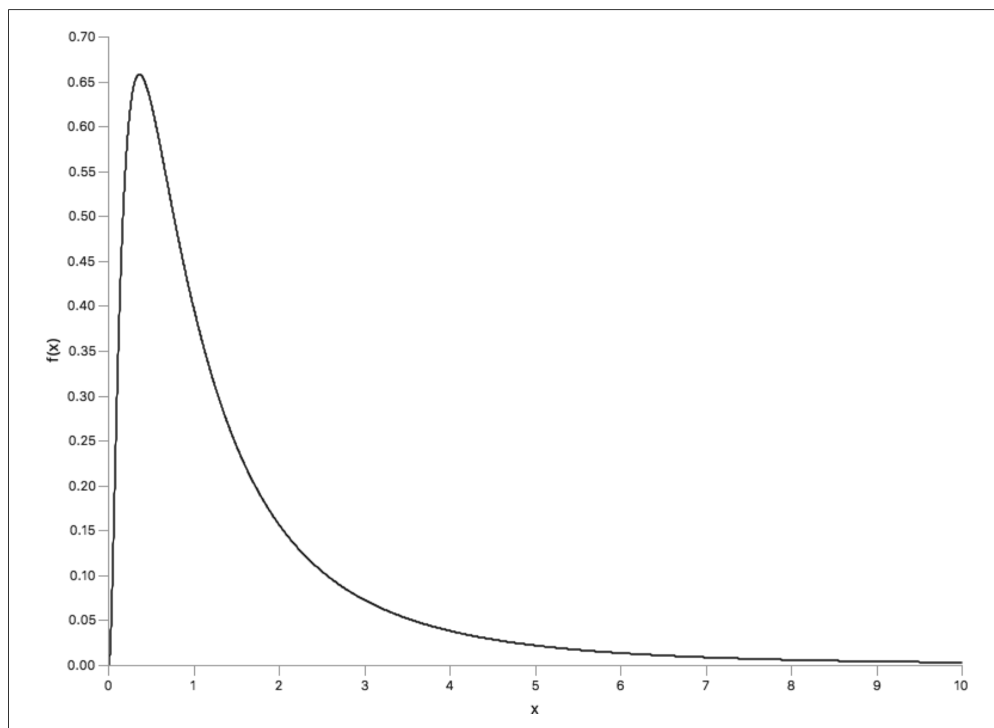


图 3-8：对数正态分布的 PDF，参数 $m=0$，$s=1$

其位置（刻度）参数 m 和形状参数 s 决定了 PDF：

$$f(x)=\frac{1}{\sqrt{2\pi}xs}\exp\left(-\frac{(\ln x-m)^2}{2s^2}\right)$$

此处，m 与 s 是对数分布变量 x 的对数 $\ln x$ 的均值与标准差。其 CDF 类似于图 3-9，其数学表达式如下：

$$F(x)=\frac{1}{2}\left[1+\mathrm{erf}\left(\frac{\ln x-m}{s\sqrt{2}}\right)\right]$$

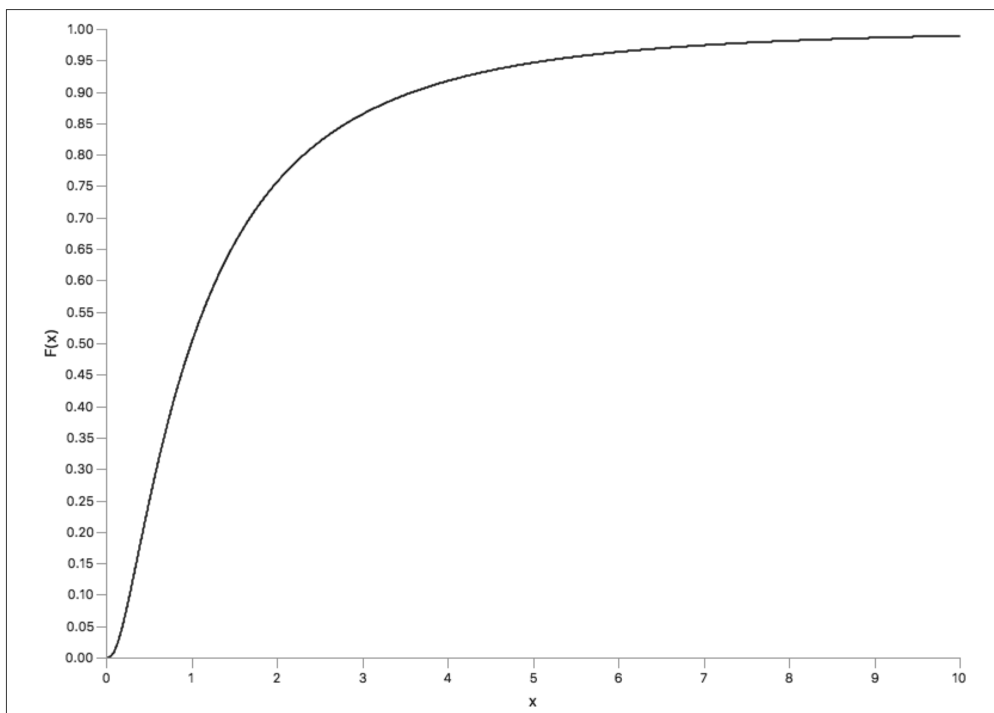


图 3-9：对数正态分布的 CDF，参数 $m = 0$，$s = 1$

与正态分布不同，m 既不是分布的平均值，也不是分布的众数（最有可能的值或者顶点）。这是因为有更多的值延展到正无穷大。变量 x 的均值与方差计算公式如下：

$$\mu = \exp(m + s^2 / 2)$$

$$\sigma^2 = \left(\exp\left(s^2\right) - 1\right)\exp\left(2m + s^2\right)$$

可以按下述方法调用对数正态分布：

```
/* 用默认的m = 0与s = 1进行初始化 */
NormalDistribution dist = new NormalDistribution();
double lowerBound = dist.getSupportLowerBound(); // 0.0
double upperBound = dist.getSupportUpperBound(); // 无穷大
double scale = dist.getScale(); // 0.0
double shape = dist.getShape(): // 1.0
double mean = dist.getNumericalMean(); // 1.649
double variance = dist.getNumericalVariance(); // 4.671
double density = dist.density(1.0); // 0.3989
double cumulativeProbability = dist.cumulativeProbability(1.0); // 0.5
double sample = dist.sample(); // 0.428
double[] samples = dist.sample(3); // {0.109, 5.284, 2.032}
```

在哪里可以看见对数正态分布？人口年龄分布，以及（有时）粒子大小的分布。注意，对数正态分布源自许多独立分布的乘法效应。

5. 经验分布

在一些场合中，虽然有数据，但是不知道数据服从什么分布。此时仍然可以根据数据近似出分布，甚至计算出概率密度、累积概率以及随机数。使用经验分布的第一步工作是，把数据搜集到若干相同大小的桶（bin）中，所有这些桶涵盖整个数据集的范围。`EmpiricalDistribution` 类可以输入 `double` 型数组，装载本地文件，或者根据 URL 装载文件。在这些情况下，数据的每行必须只有一项。

```
/* 从标准正态分布中获得2500个随机数 */
NormalDistribution nd = new NormalDistribution();
double[] data = nd.sample(2500);

// 默认的构造方法设置（桶数 = 1000）
// 尝试一下（点数/10）会更好
EmpiricalDistribution dist = new EmpiricalDistribution(25);
dist.load(data); // 也可以从文件或者URL装载数据
double lowerBound = dist.getSupportLowerBound(); // 0.5
double upperBound = dist.getSupportUpperBound(); // 10.1
double mean = dist.getNumericalMean(); // 5.48
double variance = dist.getNumericalVariance(); // 15.032
double density = dist.density(1.0); // 0.357
double cumulativeProbability = dist.cumulativeProbability(1.0); // 0.153
double sample = dist.sample(); // 例如1.396
double[] samples = dist.sample(3); // 例如[10.098, 0.7934, 9.981]
```

可以用一种称作直方图（histogram）的条形图来绘制经验分布的数据，如图 3-10 所示。

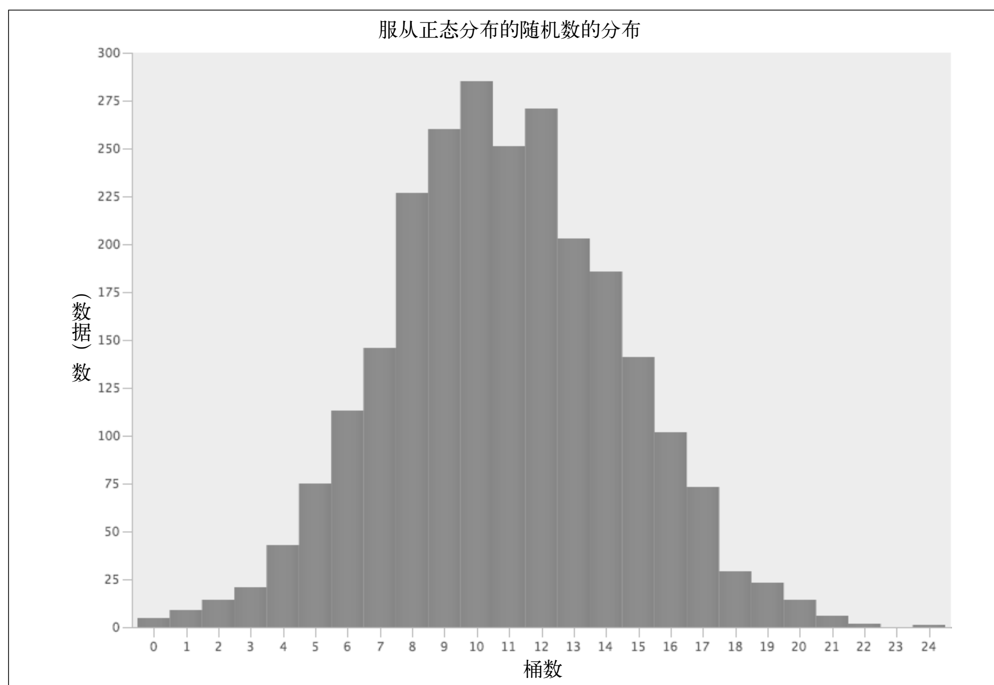


图 3-10：随机正态分布的直方图，参数 $\mu = 0$，$\sigma = 1$

直方图的代码采用了第 1 章的 BarChar 图，只是直接从 EmpiricalDistribution 实例中添加数据，该实例包含了每个桶中所有 SummaryStatistics 的 List。

```
/* 已经装载数据的现有EmpiricalDistribution对象 */
List<SummaryStatistics> ss = dist.getBinStats();
int binNum = 0;
for (SummaryStatistics s : ss) {
    /* 把桶的个数添加到XYChart.Series实例 */
    series.getData().add(new Data(Integer.toString(binNum++), s.getN()));
}
// 采用JavaFX BarChart绘制直方图
```

3.1.6 离散分布

有几种离散随机数的分布，它们只支持整型值（记作 k）。

1. 伯努利分布

伯努利分布（Bernoulli distribution）是最基本的分布，或许也是最为人所熟知的分布，因为它本质上是抛硬币。在“正面赢，反面输”的情形中，硬币有两种可能的状态：正面（$k = 1$）以及反面（$k = 0$），其中 $k = 1$ 记作成功，其概率等于 p。若硬币是均匀的，则 $p = 1/2$，即得到正面的概率和得到反面的概率相等。但是，若硬币是非均匀的，即 $p \neq 1/2$，会如何呢？此时就要用到概率质量函数（PMF），如下式所示：

$$f(k) = \begin{cases} 1-p, & k=0 \\ p, & k=1 \end{cases}$$

累积分布函数如下式所示：

$$F(k) = \begin{cases} 0, & k<0 \\ 1-p, & 0 \leqslant k < 1 \\ 1, & k \geqslant 1 \end{cases}$$

均值与方差的计算公式如下式：

$$\mu = p$$
$$\sigma^2 = p(1-p)$$

注意伯努利分布与二项分布相关，只是伯努利分布的试验次数为 $n = 1$。伯努利分布用 BinomialDistribution(1, p) 类实现，将 n 设置为 1。

```
BinomialDistribution dist = new BinomialDistribution(1, 0.5);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 1
int numTrials = dist.getNumberOfTrials(); // 1
double probSuccess = dist.getProbabilityOfSuccess(); // 0.5
```

```
double mean = dist.getNumericalMean(); // 0.5
double variance = dist.getNumericalVariance(); // 0.25
// k = 1
double probability = dist.probability(1); // 0.5
double cumulativeProbability = dist.cumulativeProbability(1); // 1.0
int sample = dist.sample(); // 例如1
int[] samples = dist.sample(3); // 例如[1, 0, 1]
```

2. 二项分布

若进行多重伯努利试验，则得到二项分布。对于 n 重伯努利试验，每次成功的概率为 p，则 k 次成功如图 3-11 所示。

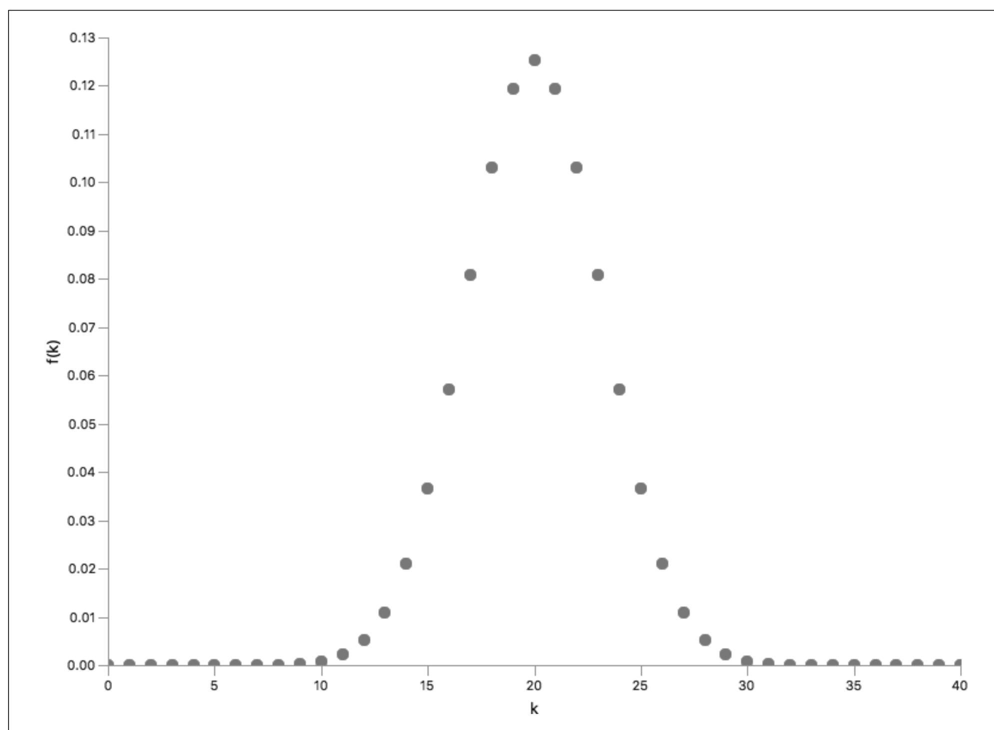


图 3-11：二项分布的 PMF，参数 $n = 40$，$p = 0.5$

概率质量函数的表达式如下：

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

CDF 如图 3-12 所示。

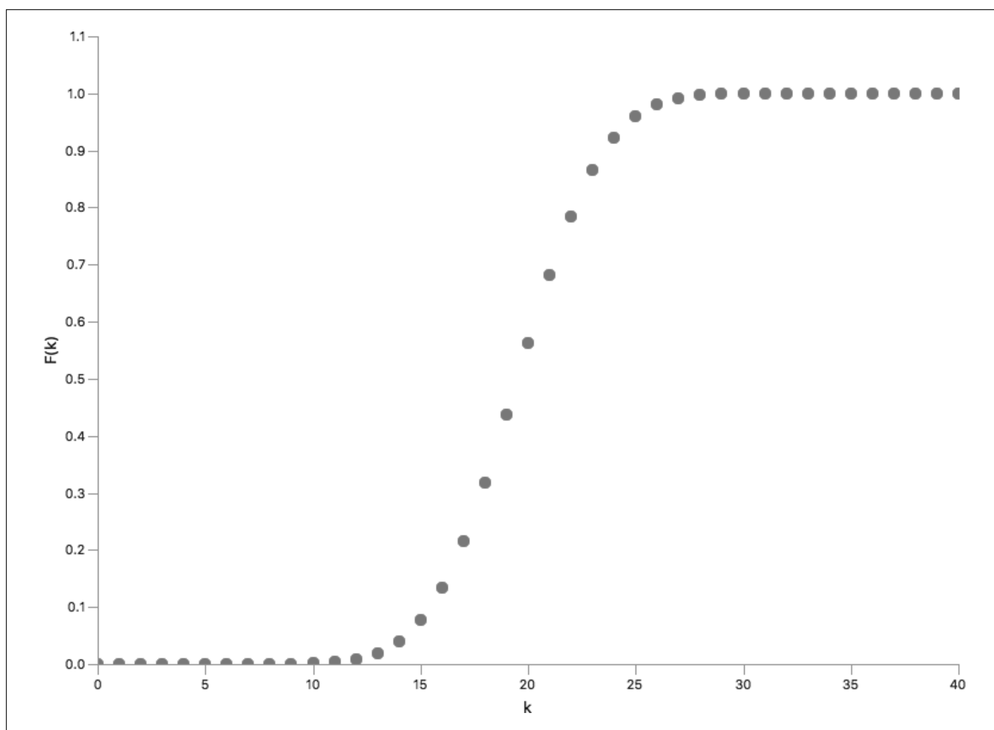


图 3-12：二项分布的 CDF，参数 $n = 40$，$p = 0.5$

CDF 的表达式如下：

$$F(k) = I_{1-p}(n-k,\ 1+k)$$

其中，I_{1-p} 是归一化的不完全 beta 函数。二项分布的均值与方差的计算公式如下：

$$\mu = np$$
$$\sigma^2 = np(1-p)$$

在 Java 中，BinomialDistribution 的构造方法有两个必需的参数：n（试验次数）和 p（一次试验中成功的概率）。

```
BinomialDistribution dist = new BinomialDistribution(10, 0.5);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 10
int numTrials = dist.getNumberOfTrials(); // 10
double probSuccess = dist.getProbabilityOfSuccess(); // 0.5
double mean = dist.getNumericalMean(); // 5.0
double variance = dist.getNumericalVariance(); // 2.5
// k = 1
double probability = dist.probability(1); // 0.00977
```

```
double cumulativeProbability = dist.cumulativeProbability(1); // 0.0107
int sample = dist.sample(); // 例如9
int[] samples = dist.sample(3); // 例如[4, 5, 4]
```

3. 泊松分布

泊松分布（Poisson distribution）常用来描述鲜有发生的离散独立事件。若事件在某个区间内以恒定比率 $\lambda > 0$ 发生，所发生事件数目是整数 $k \geqslant 0$，则所对应的 PMF 如图 3-13 所示。

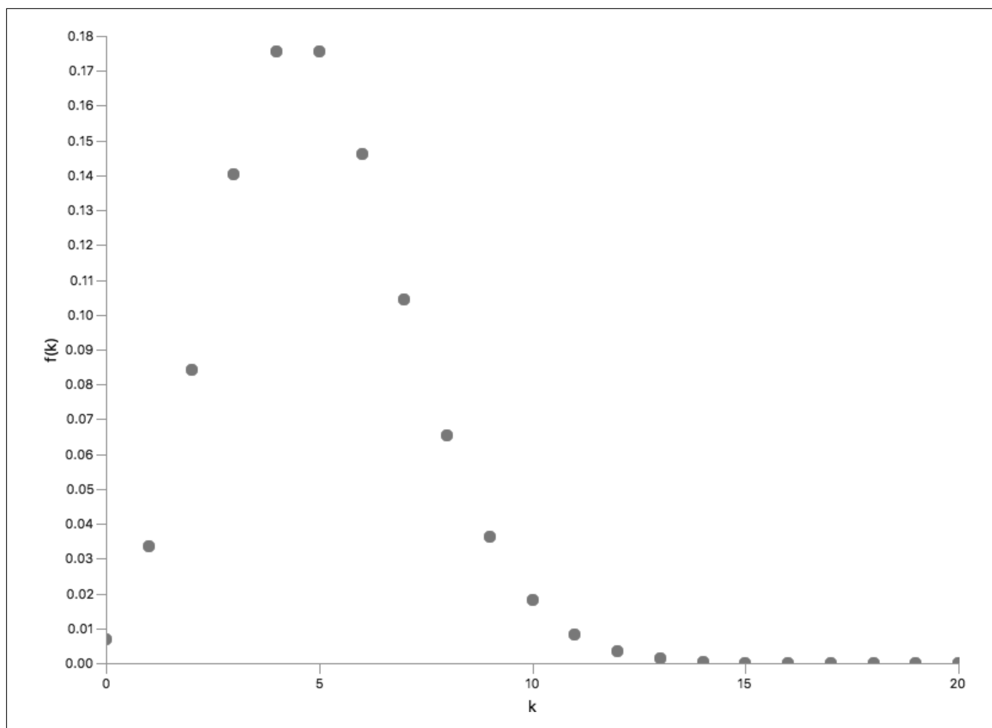


图 3-13：泊松分布的 PMF，参数 $\lambda = 5$

PMF 的数学表达式如下：

$$f(k) = \frac{\lambda^k \exp(-\lambda)}{k!}$$

图 3-14 给出了 CDF。

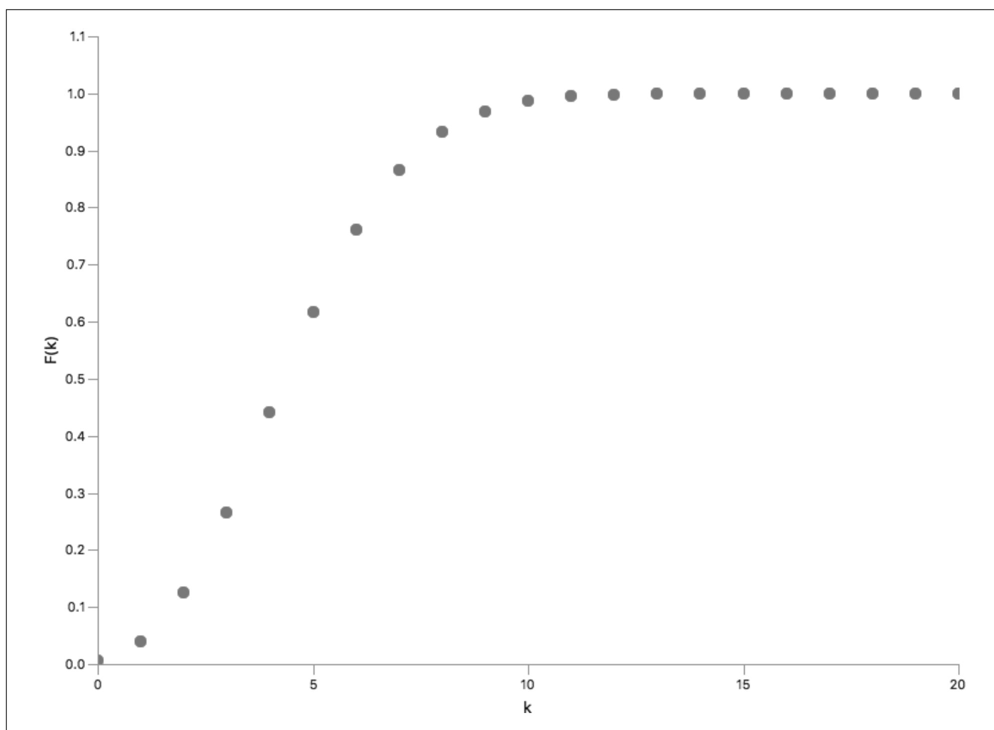


图 3-14：泊松分布的 CDF，参数 $\lambda = 5$

CDF 的数学表达式如下：

$$F(k) = \frac{\Gamma([k+1], \lambda)}{[k]!}$$

均值与方差都等于比率参数 λ，如下式所示：

$$\mu = \lambda$$
$$\sigma^2 = \lambda$$

泊松分布的实现需在构造方法中填入参数 λ，上限为 Integer.MAX（$k = 2^{32} - 1 = 2\,147\,483\,647$）。

```
PoissonDistribution dist = new PoissonDistribution(3.0);
int lowerBound = dist.getSupportLowerBound(); // 0
int upperBound = dist.getSupportUpperBound(); // 2147483647
double mean = dist.getNumericalMean(); // 3.0
double variance = dist.getNumericalVariance(); // 3.0
// k = 1
double probability = dist.probability(1); // 0.1494
double cumulativeProbability = dist.cumulativeProbability(1); // 0.1991
int sample = dist.sample(); // 例如1
int[] samples = dist.sample(3); // 例如[2, 4, 1]
```

3.2　数据集的特征

一旦有了数据集，首先要做的是理解数据的特征。应当知道的内容包括：数值极限，是否存在任何异常值，以及数据形状是否类似于已知分布函数中的一种。即使对数据潜在的分布一无所知，也仍然可以检查两个数据集是否来自同一（未知的）分布。也可以通过协方差 / 相关系数来检查每对变量的相关性（或不相关性）。若变量 x 伴随着响应 y，则可以通过检查线性回归来判断 x 与 y 之间是否存在着最基本的关系。本节中的绝大多数类最适合于小的静态数据集，可以完全装入内存，因为这些类中的大多数方法都依赖于内存中的数据。下一节中，所处理的数据将大（或者不方便）到不适合装载到内存中。

3.2.1　矩的计算

前一节中讨论了统计矩，其中所给出的计算矩及其各种统计结果的公式针对已知概率分布函数 $f(x)$ 的情形。处理实际数据时，通常不知道 $f(x)$，因此必须用数值方法估计矩。矩的计算有另一个关键特征：稳定性。当遇到极值时，对统计量的估计可以导致数值误差。采用更新矩的方法，可以避免数值的不准确。

1. 样本矩

对于实际数据，我们可能并不知道其真实的统计分布函数，但可以估计中心矩：

$$m_k = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^k$$

此处的估计均值 $m_1 = \overline{x}$，其计算公式如下：

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

2. 矩的更新

去掉 $1/n$ 的因子，估计中心矩的公式变为下式：

$$M_k = \sum_{i=1}^{n}(x_i - \overline{x})^k$$

在这种特定的形式中，未归一化的矩可以通过直接计算分解为多个部分。这样做有很大的优势，即对未归一化的矩按不同部分进行计算，从而可以在不同的进程中计算，甚至是在完全不同的机器上计算。随后可以把它们拼接在一起。另一优势是这种计算方法对极值不太敏感。两个分块的非归一化中心矩的组合公式如下：

$$M_k = M_{k,1} + M_{k,2} + \sum_{j=1}^{k-2}\binom{j}{k}\left[\left(-\frac{n_2}{n}\right)^j M_{k-j,1} + \left(\frac{n_1}{n}\right)^j M_{k-j,2}\right]\delta_{2,1}^j + \left(\frac{n_1 n_2}{n}\delta_{2,1}\right)^k\left[\frac{1}{n_2^{k-1}} - \left(\frac{-1}{n_1}\right)^{k-1}\right]$$

$\delta_{2,1} = \overline{x}_2 - \overline{x}_1$ 是两个数据分块的均值之差。当然，因为上面的公式只适用于 $k > 1$，所以也需要合并均值的方法。给定任何两个数据分块，已知各自均值与数据点个数，总数据点个数为 $n = n_1 + n_2$，那么按下式计算的均值是稳定的：

$$\overline{x} = \overline{x}_1 + n_2 \frac{\delta_{2,1}}{n}$$

若其中某个数据分块只有单个点 x，那么非归一化中心矩的组合公式可以简化为：

$$M_k = M_{k,1} + \sum_{j=1}^{k-2} \binom{j}{k} M_{k-j,1} \left(\frac{-\delta}{n}\right)^j + \left(\frac{n-1}{n}\delta\right)^k \left[1 - \left(\frac{-1}{n-1}\right)^{k-1}\right]$$

此处，$\delta = x - \overline{x}_1$ 是新增值 x 与现有数据分块均值之差。

下一节中将看到如何用这些公式稳定地计算重要的统计结果。在分布式计算应用中，若希望把统计计算分为多个部分，则这些公式是必要的。另一个有用的应用是无存储的计算，它不是针对整个数据数组进行计算，而是逐步地记录矩，并对它们进行递增更新。

3.2.2 描述性统计

对 `DescriptiveStatistics` 类进行实例化，但不设置任何参数，以便以后再添加值或者赋予 `double` 型数组（以后仍然可以再填充值）。可以使用 `StatUtils` 类的静态方法，虽然也没有什么错误，但这不像是 Java 的用法，使用 `DescriptiveStatistics` 类可能更明智。本节中的一些公式是不稳定的，下一节中描述的公式会更稳定。的确，那些方法中的一些也用在描述性统计方法中。表 3-1 中列出了 Anscombe 的 4 组数据，供本章进一步分析所用。

表3-1：Anscombe的4组数据

x1	y1	x2	y2	x3	y3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

然后，可以用这些数据集创建 DescriptiveStatistics 类。

```
/* Anscombe数据集中y1的统计值 */
DescriptiveStatistics descriptiveStatistics = new DescriptiveStatistics();
descriptiveStatistics.addValue(8.04);
descriptiveStatistics.addValue(6.95);
// 继续添加y1的值
```

然而，你可能已经有自己所需的所有数据，或者它只是初始集，将在随后添加数据。如果需要，就可以用 ds.addValue(double value) 方法一直添加更多的值。此时，可以通过调用这个方法或者直接打印这个类来给出统计报表：

```
System.out.println(descriptiveStatistics);
```

这将产生下面的结果：

```
DescriptiveStatistics:
n: 11
min: 4.26
max: 10.84
mean: 7.500909090909091
std dev: 2.031568135925815
median: 7.58
skewness: -0.06503554811157437
kurtosis: -0.5348977343727395
```

所有这些（以及更多的）量可以通过其相应的获取值方法获得，举例如下。

1. 计数

最简单的统计量是数据集中数据点的个数：

```
long count = descriptiveStatistics.getN();
```

2. 总和

也可以得到所有值的总和：

```
double sum = descriptiveStatistics.getSum();
```

3. 最小值

用这个方法获得数据集的最小值：

```
double min = descriptiveStatistics.getMin();
```

4. 最大值

用这个方法获得数据集的最大值：

```
double max = descriptiveStatistics.getMax();
```

5. 均值

可以直接计算样本的平均值，即均值：

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

然而，这种计算方法对于极值是敏感的。给定 $\delta = x - \overline{x}$ ，对于每个新添的值 x，可以对均值进行更新：

$$\overline{x} = \overline{x}_1 + \frac{x - \overline{x}_1}{n}$$

调用 `getMean()` 方法时，Commons Math 采用更新均值的计算公式：

```
double mean = descriptiveStatistics.getMean();
```

6. 中位数

中位数（median）是有序（升序）数据集的中间值，其优点是使得极值的问题最小化。尽管 Apache Commons Math 中没有直接计算中位数的方法，但是还是容易计算中位数的。当数组长度是偶数时，取两个中间元素的均值；否则，只要返回数组最中间的元素则可。

```
// 对存储的值进行排序
double[] sorted = descriptiveStatistics.getSortedValues();
int n = sorted.length;
double median = (n % 2 == 0) ? (sorted[n/2-1]+sorted[n/2])/2.0 : sorted[n/2];
```

7. 众数

众数（mode）是最可能出现的值。若值是 `double` 型，则众数的概念就没有意义，因为每个值只能出现一次。很明显，也有例外，例如数字中有许多零，或者数据集大但是数值精度小（例如两位小数）时。于是，众数有两个用途：若所考虑的变量是离散的（整数），则众数是有用的，就像在 Anscombe 的 4 组数据的第 4 组那样。此外，若已经从经验分布中创建了桶，众数就是最大的桶。然而，需要考虑数据中有噪声的可能性，从而桶计数器会错误地把异常值标记为众数。`StatUtils` 类包含了几个对统计有用的静态方法，此处使用其求众数的方法。

```
// 若出现次数最多的值只有一个，则将其存储在mode[0]中
// 若出现次数最多的值多于一个，则将这些值按升序存储在mode数组中
double[] mode = StatUtils.mode(x4);
// mode[0] = 8.0
double[] test = {1.0, 2.0, 2.0, 3.0, 3.0, 4.0}
// mode[0] = 2.0
// mode[1] = 3.0
```

8. 方差

方差（variance）是一种反映数据分布有多广的度量，它是永远大于或等于零的实数值。若所有的 x 值都相等，则方差是零。相反，数据点分布范围越大，对应的方差就越大。已知数据点总体的方差等于均值的二阶中心矩，表达式如下：

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

然而，大多数时候我们并没有全部数据，仅是从更大的（可能是未知的）数据集中抽取了样本，因此需要对这个偏差进行修正：

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

这种形式称作**样本方差**（sample variance），也是最常用的方差。注意，样本方差可以用非归一化的二阶矩表示为：

$$s^2 = \frac{1}{n-1}M_2$$

就像均值计算中根据新数据点 x 以及已有均值 $\overline{x}_1$ 对均值 $\overline{x}$ 进行更新那样，Commons Math 根据新数据点 x 以及已有均值 $\overline{x}_1$ 采用非归一化的二阶矩更新公式计算方差：

$$M_2 = M_{2,1} + \frac{n-1}{n}\delta^2$$

此处 $\delta = x - \overline{x}_1$ 。

因为数据通常是从一些更大的、可能是未知的数据集中选取的样本，所以大多数时候，当需要方差时，所求的方差指的是纠偏的样本方差：

```
double variance = descriptiveStatistics.getVariance();
```

然而如果需要，获得总体方差的方法也很直接：

```
double populationVariance = descriptiveStatistics.getPopulationVariance();
```

9. 标准差

方差可视化起来比较困难，因为它是 x^2 的量级，相对于均值而言，通常是个大的数值。对方差取平方根，定义该值为标准差 s，它具有与变量和均值单位相同的优点。因此，采用 $\mu \pm \sigma$ 之类的值是有帮助的，它表明数据偏离均值的程度。可以用下面的方法显式地计算标准差：

$$s=\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\overline{x})^2}$$

然而在实际中，采用更新公式来计算样本方差。当需要时，返回样本方差的平方根作为标准差：

```
double standardDeviation = descriptiveStatistics.getStandardDeviation();
```

若需要总体标准差，则可以对总体方差求平方根直接计算。

10. 均值的误差

尽管经常假设标准差是均值的误差，实际却不是这样。标准差描述了数据是如何围绕平均值分布的。通过下式，可以利用标准差计算均值本身的准确率 s_x。

$$s_x=\frac{s}{\sqrt{n}}$$

也可以用简单一点的 Java 方法计算。

```
double meanErr = descriptiveStatistics.getStandardDeviation() /
    Math.sqrt(descriptiveStatistics.getN());
```

11. 偏度

偏度（skewness）用来度量数据分布的非对称性，它可以是正实数或负实数。正的偏度表明大多数值倾向原点（$x = 0$）；负的偏度表明值向远处分布（向右）；偏度为 0 表明数据完美地分布在数据分布峰值的两侧。偏度可以用下面的表达式显式地计算：

$$\omega=\frac{1}{(n-1)(n-2)}\sum_{i=1}^{n}\left(\frac{x_i-\overline{x}}{s}\right)^3$$

然而，关于偏度更稳定的计算是通过更新三阶中心矩得到的。

$$M_3=M_{3,1}-3M_{2,1}\frac{\delta}{n}+(n-1)(n-2)\frac{\delta^3}{n^2}$$

然后，根据需要，可以计算偏度。

$$\omega=\frac{1}{(n-1)(n-2)}\frac{M_3}{s^3}$$

Commons Math 的实现对存储在内存中的数据集进行迭代，递增地更新 M_3，然后进行纠偏，并返回偏度。

```
double skewness = descriptiveStatistics.getSkewness();
```

12. 峰度

峰度是关于数据分布尾部的一种度量。样本峰度估计与关于均值的四阶中心矩相关，计算公式如下：

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \sum_{i=1}^{n} \left(\frac{x_i - \overline{x}}{s} \right)^4$$

可以简化为下式：

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \frac{M_4}{s^4}$$

峰度等于 0 或接近于 0 时，表明数据分布是极其窄的。随着峰度的增大，极值会出现在长尾的地方。不过，我们通常想把峰度表示为与正态分布相关（正态分布的峰度 =3）。于是可以从中减去 3，并将新量称作**超值峰度**，但实际上大多数人把超值峰度直接称作峰度。在这个定义中[1]，$\kappa = 0$ 表示数据与正态分布有相同的峰值与尾部形状。峰度[2] 大于 3 时，分布称作**尖峰态的**（leptokurtic），其尾部比正态分布要宽。当 κ 小于 3 时[3]，分布称作**低峰态的**（platykurtic），其尾部值较少（比正态分布要少）。超值峰度计算公式如下式：

$$\kappa = \frac{(n+1)}{(n-1)(n-2)(n-3)} \frac{M_4}{s^4} - \frac{3(n-1)^2}{(n-2)(n-3)}$$

就像方差与偏度那样，峰度通过对非归一化的四阶中心矩进行更新来计算。对于添加到计算中的每个点，只要点的数目 $n \geqslant 4$，就可以用下面的公式对 M_4 更新。在任何点上，峰度可以从 M_4 的当前值计算得出。

$$M_4 = M_{4,1} - 4M_{3,1}\frac{\delta}{n} + 6M_{2,1}\left(\frac{\delta}{n}\right)^2 + (n-1)(n^2-3n+3)\frac{\delta^4}{n^3}$$

采用 `getKurtosis()` 方法，默认返回值是根据超值峰度定义计算得出的。这可以通过实现 `org.apache.commons.math3.stat.descriptive.moment.Kurtosis` 类来核实，在 `getKurtosis()` 方法中调用该类。

```
double kurtosis = descriptiveStatistics.getKurtosis();
```

3.2.3 多元统计

至此我们已经讨论了一次只有单个变量的情形。`DescriptiveStatistics` 类仅处理一维

注 1：此处指超值峰度。——译者注

注 2：此处指的峰度的原始定义，不是超值峰度。——译者注

注 3：此处的 κ 采用的是峰度的原始定义，不是超值峰度。——译者注

数据。然而，通常有几个维度，有几百个维度也是很常见的。有两种选择，第一种是用 MultivariateStatisticalSummary 类，在下一节描述它。若可以不使用偏度与峰度，则这将是最好的选择。若确实需要所有统计量，则最佳选择是实现关于 DescriptiveStatistics 对象的 Collection 实例。首先确定想要了解什么。例如，在 Anscombe 的 4 组数据中，可以用下面的方法获取一元统计值。

```
DescriptiveStatistics descriptiveStatisticsX1 = new DescriptiveStatistics(x1);
DescriptiveStatistics descriptiveStatisticsX2 = new DescriptiveStatistics(x2);
...
List<DescriptiveStatistics> dsList = new ArrayList<>();
dsList.add(descriptiveStatisticsX1);
dsList.add(descriptiveStatisticsX2);
...
```

然后，可以对 List 进行迭代，获取统计量或者原始数据。

```
for(DescriptiveStatistics ds : dsList) {

    double[] data = ds.getValues();
    // 利用数据进行操作，或者

    double kurtosis = ds.getKurtosis();
    // 利用峰度进行操作
}
```

若数据集更加复杂，且随后的分析中需要获取数据的特定列，则可以使用 Map。

```
DescriptiveStatistics descriptiveStatisticsX1 = new DescriptiveStatistics(x1);
DescriptiveStatistics descriptiveStatisticsY1 = new DescriptiveStatistics(y1);
DescriptiveStatistics descriptiveStatisticsX2 = new DescriptiveStatistics(x2);

Map<String, DescriptiveStatistics> dsMap = new HashMap<>();
dsMap.put("x1", descriptiveStatisticsX1);
dsMap.put("y1", descriptiveStatisticsY1);
dsMap.put("x2", descriptiveStatisticsX2);
```

当然，现在可以很容易地通过键获取特定的量或者数据集。

```
double x1Skewness = dsMap.get("x1").getSkewness();
double[] x1Values = dsMap.get("x1").getValues();
```

当维数很多时，这将变得困难。但若数据已经存储在多维数组（或矩阵）中时，则可以简化这个过程，可以对列索引进行遍历。同样，由于可能已经把数据存储在数据容器类的 List 或 Map 中，因此把构造关于 DescriptiveStatistics 对象的多维 Collection 的过程自动化是直截了当的。如果已经有数据字典（关于变量名及其属性的列表），那么根据它进行迭代是特别有效的。若有同一种类型的高维数值数据，它已经在 double 型数组形式的矩阵中存在，则使用下一节的 MultivariateSummaryStatistics 类可能会更容易。

3.2.4 协方差与相关系数

协方差矩阵与相关系数矩阵是对称的 $m \times m$ 方阵，其维数 m 等于原始数据集的列数。

1. 协方差

协方差相当于二维变量的方差，它刻画两个变量与各自均值差距的总体情况，其计算公式如下：

$$\sigma_{i,j} = \frac{1}{n-1}\sum_{k=1}^{n}\left(x_{k,i} - \overline{x}_i\right)\left(x_{k,j} - \overline{x}_j\right)$$

就像在一维变量中样本统计矩的情形一样，注意到下面的量：

$$C_2 = \sum_{k=1}^{n}\left(x_{k,i} - \overline{x}_i\right)\left(x_{k,j} - \overline{x}_j\right)$$

在给定均值以及现有维数时，可以表示为一对变量 x_i 与 x_j 协矩的增量更新。

$$C_2 = C_{2,1} + C_{2,2} + \frac{n_1 n_2}{n}\left(\overline{x}_{i_1} - \overline{x}_{i_2}\right)\left(\overline{x}_{j_1} - \overline{x}_{j_2}\right)$$

然后，协方差可以在任一点进行计算：

$$\sigma_{i,j} = \frac{1}{n-1} C_2$$

计算协方差的代码如下：

```
Covariance cov = new Covariance();

/* 采用Anscombe的4组数据作为示例 */
double cov1 = cov.covariance(x1, y1); // 5.501
double cov2 = cov.covariance(x2, y2); // 5.499
double cov3 = cov.covariance(x3, y3); // 5.497
double cov4 = cov.covariance(x4, y5); // 5.499
```

若数据已经存储在二维的 `double` 型数组或者 `RealMatrix` 实例中，则可以把它们直接传递给构造方法，就像下面这样：

```
//double[][] myData或者RealMatrix myData
Covariance covObj = new Covariance(myData);
//cov包含协方差，可以用RealMatrix.get(i,j)获取元素以对cov进行访问
RealMatrix cov = covObj.getCovarianceMatrix();
```

注意协方差矩阵 $\sigma_{i,j}$ 的对角线正是列 i 的方差，因此协方差矩阵对角线的平方根是每维数据的标准差。因为总体均值通常未知，所以采用样本均值的有偏协方差。若确实知道总体

均值，则将可以用无偏的修正因子 $1/n$ 计算无偏的协方差。

$$\sigma_{i,j} = \frac{1}{n} C_2$$

2. 皮尔逊相关系数

皮尔逊相关系数与协方差的关系如下式，它用来度量两个变量一同变化的可能性有多大：

$$\rho_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j}$$

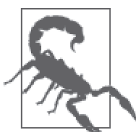
相关系数的取值范围是 −1~1，其中 1 表示两个变量几乎相同，−1 表示它们是相反的。在 Java 中同样有两种选项，用默认构造方法得到下列代码：

```
PearsonsCorrelation corr = new PearsonsCorrelation();

/* 采用Anscombe的4组数据作为示例 */
double corr1 = corr.correlation(x1, y1)); // 0.816
double corr2 = corr.correlation(x2, y2)); // 0.816
double corr3 = corr.correlation(x3, y3)); // 0.816
double corr4 = corr.correlation(x4, y4)); // 0.816
```

然而，如果已有数据或者 `Covariance` 实例，那么可以采用下面的方法：

```
// 已有Covariance实例cov
PearsonsCorrelation corrObj = new PearsonsCorrelation(cov);
// double[][] myData或者RealMatrix myData
PearsonsCorrelation corrObj = new PearsonsCorrelation(myData);
// 用RealMatrix.get(i,j)获取元素
RealMatrix corr = corrObj.getCorrelationMatrix();
```



相关性并不意味着因果关系。统计学中的危险之一是对相关性的不当解读。若两个变量的相关性高，则倾向于假定一个变量导致了另一个变量。但是并非如此。事实上，所能假定的是可以拒绝变量之间毫无关联的想法。应当把相关性看作一种幸运的巧合，而不是所研究的系统底层行为的根本基础。

3.2.5　回归

通常人们想要找到变量 $\boldsymbol{X}$ 以及它们的响应 y 之间的关系，也就是试图寻找一系列的值 β，使得 $y = \boldsymbol{X}\hat{\beta}$。最后，我们需要 3 种量：参数、它们的误差，以及表示拟合好坏的统计值 R^2。

1. 简单回归

若 $\boldsymbol{X}$ 是一维的，则问题是人们熟知的线性方程 $y = \hat{\alpha} + \hat{\beta}x$，该问题可以归类为简单回归。通过计算 x 的方差 σ_x^2，以及 x 与 y 之间的协方差 $\sigma_{x,y}$，可以估计斜率。

$$\hat{\beta} = \frac{\sigma_{x,y}}{\sigma_x^2}$$

然后，采用斜率以及 x 与 y 的均值，可以估计截距。

$$\hat{\alpha} = \overline{y} - \hat{\beta}\overline{x}$$

Java 代码采用了 `SimpleRegression` 类。

```
SimpleRegression rg = new SimpleRegression();

/* Anscombe的4组数据中的x1与y1的x-y对 */
double[][] xyData = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

rg.addData(xyData);

/* 获得回归的结果 */
double alpha = rg.getIntercept(); // 3.0
double alpha_err = rg.getInterceptStdErr(); // 1.12
double beta = rg.getSlope(); // 0.5
double beta_err = rg.getSlopeStdErr(); // 0.12
double r2 = rg.getRSquare(); // 0.67
```

于是，可以将这些结果解析为 $y = 3.0 + 0.5x$，或者更加具体地解析为 $y = (3.0 \pm 1.12) + (0.5 \pm 0.12)x$。在多大程度上可以信任这个模型？ $R^2 = 0.67$ 是个相当不错的拟合，但是越接近于理想的 $R^2 = 1.0$ 越好。注意，若对 Anscombe 的 4 组数据中另外 3 个数据集进行相同的回归分析，则可以得到同样的参数、误差以及 R^2。这个结果尽管令人困惑，但意义深远。显然，虽然 4 个数据集看上去如此不同，但是它们的线性拟合（叠放在上面的蓝色直线）是相同的。尽管对第一组数据来说，线性回归是理解数据的一种强大且简单的方法，但是对第 2 组数据来说，此处对 x 进行线性回归可能并不合适。在第 3 组数据中，线性回归可能是正确的工具，但是要注意（移除）那个看上去像异常值的数据点。在第 4 组数据中，回归模型恐怕根本不适用。这表明，如果盲目地用一种分析方法处理数据之后，只根据一些参数就认为模型是正确的，那就很容易使自己被愚弄。

2. 多元回归

解决上述问题有多种方法，但是最常见且可能最有用的是普通最小二乘（OLS）法，其解用线性代数表示如下：

$$\hat{\beta} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}y$$

Apache Commons Math 中的 `OLSMultipleLinearRegression` 类对 QR 分解做了适当的封装。该实现也提供了除 QR 分解之外的额外功能，你会发现这些功能是有用的。其中，特别要

指出的是，β 的方差－协方差矩阵如下，其中矩阵 $\boldsymbol{R}$ 是从 QR 分解中得出的。

$$\sigma_{\hat{\beta}}^2 = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1} = \left(\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}\right)^{-1}$$

在这种情况下，$\boldsymbol{R}$ 必须截断至 β 的维度。给定拟合余数 $\hat{\epsilon} = y - X\hat{\beta}$，可以计算误差的方差 $s_{\text{err}}^2 = \hat{\epsilon}^{\mathrm{T}}\hat{\epsilon} / (n-p)$，其中 n 与 p 分别是 $\boldsymbol{X}$ 行与列的数目，矩阵 $\sigma_{\hat{\beta}}^2$ 的对角线值的平方根乘以常数 s_{err} 给出了关于拟合参数的误差估计。

$$\delta\hat{\beta}_i = s_{\text{err}}\sqrt{\sigma_{\hat{\beta}_{i,i}}^2}$$

Apache Commons Math 实现的普通最小二乘回归利用了线性代数中的 QR 分解。示例代码中的方法是关于矩阵的几个标准操作的适当封装。注意默认包括截距项，对应值在估计参数的第一个位置。

```
double[][] xNData = {{0, 0.5}, {1, 1.2}, {2, 2.5}, {3, 3.6}};
double[] yNData = {-1, 0.2, 0.9, 2.1};
// 默认会包含截距项
OLSMultipleLinearRegression mr = new OLSMultipleLinearRegression();
/* 注意：与其他的类/方法不同，此处y与x的位置是倒过来的 */
mr.newSampleData(yNData, xNData);
double[] beta = mr.estimateRegressionParameters();
// [-0.7499, 1.588, -0.5555]
double[] errs = mr.estimateRegressionParametersStandardErrors();
// [0.2635, 0.6626, 0.6211]
double r2 = mr.calculateRSquared();
// 0.9945
```

线性回归是个庞大的课题，它有多种变形。由于其变形太多了，所以此处不能对其全部进行讨论。然而需要注意的是，只有 $\boldsymbol{X}$ 与 y 之间确实是线性关系时，这些方法才具有意义。自然界充满了非线性关系，第 5 章将讨论处理非线性关系的更多方式。

3.3　处理大数据集

当数据如此之大，以至于将其存储在内存中是低效的（或者根本就装不下）时，需要另一种计算统计结果的方法。`DescriptiveStatistics` 等类在实例化期间，将所有数据都存储在内存中。不过另一种处理这个问题的方式是只保存非归一化的统计矩，并一次用一个数据点对这些非归一化的统计矩进行更新。更新之后，就把已经使用过的数据点丢弃。Apache Commons Math 有两个这样的类：`SummaryStatistics` 与 `MultivariateSummaryStatistics`。

也可以对非归一化矩并行累加，从而使这种方法的有效性得以增强。可以把数据进行分块，当一次添加一个值时，记录每个分块的矩。最后，可以把所有这些矩进行归并，从而得到概要统计量。Apache Commons Math 的 `AggregateSummaryStatistics` 类可以完成这种操作。很容易想象，TB 量级的数据分布在大集群中，其中每个节点负责更新统计矩。当

作业结束时，就可以用简单的计算完成矩的归并，任务也就结束了。

通常，数据集 $\boldsymbol{X}$ 可以划分为 k 个较小数据集：$\boldsymbol{X}_1, \boldsymbol{X}_2, \cdots, \boldsymbol{X}_k$。理想情况下，可以对每个子集 $\boldsymbol{X}_i$ 进行各种计算，再对这些结果归并，从而得到所需要的关于 $\boldsymbol{X}$ 的统计量。例如，若想要统计 $\boldsymbol{X}$ 中数据点的个数，则可以统计每个子集中数据点的个数，再把这些结果累加，从而得到总体数据点的个数。

$$n = n_1 + n_2 + \cdots + n_k$$

无论各个子集是在同一台机器的不同线程中计算，还是在完全不同的机器上计算，上式都是成立的。

因此，若计算了数据点的个数，再计算每个子集各自的值之和（并对其进行记录），则随后可以利用这些信息，以分布式的方法计算 $\boldsymbol{X}$ 的均值。

$$\overline{x} = \frac{\sum_{x \in X_1} x + \sum_{x \in X_2} x + \cdots + \sum_{x \in X_k} x}{n_1 + n_2 + \cdots + n_k}$$

最简单的情形可以是只做成对操作的计算，因为任何数目的操作都可以归结为这种方式。例如，$\boldsymbol{X} = (\boldsymbol{X}_1 + \boldsymbol{X}_2) + (\boldsymbol{X}_3 + \boldsymbol{X}_4)$ 是 3 次成对操作的组合。于是，有 3 种成对算法的通用情形：第 1 种，对两个分块归并时，每个分块数据点数 $n_i > 1$；第 2 种，只有一个分块的数据点数 $n_i > 1$，而其余分块只有单个数据，即 $n_i = 1$；第 3 种，所有分块都只包含单个数据。

3.3.1　累积统计

在前一章中已经讨论了统计量是如何更新的。读者或许会想到，可以在不同时间及不同机器上计算并存储（非归一化的）矩，当方便时对其进行更新。只要记录数据点数目，以及所有相关的统计矩，就可以在任何时候重新获取它们并用新的数据点集合更新它们。`DescriptiveStatistics` 类在一系列计算中存储所有数据，并进行这些更新。`SummaryStatistics` 类（以及 `MultivariateSummaryStatistics` 类）并不存储输入给它们的任何数据。相反，这些类只存储相关的 n、M_1 以及 M_2。对于巨大的数据集，当需要均值或标准差等统计量时，这是一种记录统计量的有效方式，而不会造成庞大的存储开销或者处理能力需求。

```
SummaryStatistics ss = new SummaryStatistics();

/* 该类并不存储数据，因此已经优化为一次仅处理一个值 */
ss.addValue(1.0);
ss.addValue(11.0);
ss.addValue(5.0);

/* 打印报告 */
System.out.println(ss);
```

与 DescriptiveStatistics 类一样，SummaryStatistics 类也有 toString() 方法，该方法可以打印具有良好格式的报告。

```
SummaryStatistics:
n: 3
min: 1.0
max: 11.0
sum: 17.0
mean: 5.666666666666667
geometric mean: 3.8029524607613916
variance: 25.333333333333332
population variance: 16.88888888888889
second moment: 50.666666666666664
sum of squares: 147.0
standard deviation: 5.033222956847166
sum of logs: 4.007333185232471
```

对于多元统计，MultivariateSummaryStatistics 类正好类似于其一维情形的对应类。为了实例化这个类，必须指定变量的维数（即数据集中列的数目），并指明输入数据是否是样本。通常这个选项应当设置为 true，但要注意，若忘记设置，则默认值是 false，这将带来不良的后果。MultivariateSummaryStatistics 类包含了记录每组变量之间协方差的方法。把构造方法的参数 isCovarianceBiasedCorrected 设置为 true，将对协方差采用有偏的校正因子。

```
MultivariateSummaryStatistics mss = new MultivariateSummaryStatistics(3, true);

/* 数据可以是二维数组、矩阵，或者是含有double型数组数据域的类 */
double[] x1 = {1.0, 2.0, 1.2};
double[] x2 = {11.0, 21.0, 10.2};
double[] x3 = {5.0, 7.0, 0.2};

/* 该类并不存储数据，因此已经优化为一次仅处理一个值 */
mss.addValue(x1);
mss.addValue(x2);
mss.addValue(x3);

/* 打印报告 */
System.out.println(mss);
```

正如在 SummaryStatistics 类中，该类可以打印有格式的报告，不过此处增加了协方差矩阵。

```
MultivariateSummaryStatistics:
n: 3
min: 1.0, 2.0, 0.2
max: 11.0, 21.0, 10.2
mean: 5.666666666666667, 10.0, 3.866666666666667
geometric mean: 3.8029524607613916, 6.649399761150975, 1.3477328201610665
sum of squares: 147.0, 494.0, 105.52
sum of logarithms: 4.007333185232471, 5.683579767338681, 0.8952713646500794
standard deviation: 5.033222956847166, 9.848857801796104, 5.507570547286103
covariance: Array2DRowRealMatrix{{25.3333333333,49.0,24.3333333333},
{49.0,97.0,51.0},{24.3333333333,51.0,30.3333333333}}
```

当然，每个量都可以通过其获取值方法获得。

```
int d = mss.getDimension();
long n = mss.getDimension();
double[] min = mss.getMin();
double[] max = mss.getMax();
double[] mean = mss.getMean();
double[] std = mss.getStandardDeviation();
RealMatrix cov = mss.getCovariance();
```



此时，在 SummaryStatistics 类与 MultivariateSummaryStatistics 类中，还没有计算三阶矩与四阶矩，因此暂时还得不到偏度与峰度，它们正在准备中。

3.3.2 统计结果的归并

也可以对非归一化的统计矩以及协矩进行归并。在并行处理数据分块，且所有子过程结束后，对结果进行归并。

这种任务采用 AggregateSummaryStatistics 类来处理。一般来说，统计矩随着阶数递增而传播。换句话说，为了计算三阶矩 M_3，将需要 M_2 与 M_1。因此，必须先计算并更新最高阶的矩，然后向下进行。

例如，如前所述，计算 $\delta_{2,1}$ 之后，用下式更新 M_4：

$$M_4 = M_{4,1} + M_{4,2} + n_1 n_2 \left(n_1^2 - n_1 n_2 + n_2^2\right)\frac{\delta_{2,1}^4}{n^3} + 6\left(n_1^2 M_{2,2} - n_2^2 M_{2,1}\right)\frac{\delta_{2,1}^2}{n^2} + 4\left(n_1 M_{3,2} - n_2 M_{3,1}\right)\frac{\delta_{2,1}}{n}$$

然后用下式更新 M_3：

$$M_3 = M_{3,1} + M_{3,2} + n_1 n_2 \left(n_1 - n_2\right)\frac{\delta_{2,1}^3}{n^2} + 3\left(n_1 M_{2,2} - n_2 M_{2,1}\right)\frac{\delta_{2,1}}{n}$$

接下来，用下式更新 M_2：

$$M_2 = M_{2,1} + M_{2,2} + n_1 n_2 \frac{\delta_{2,1}^2}{n}$$

最后，用下式更新均值：

$$\bar{x} = \bar{x}_1 + n_2 \frac{\delta_{2,1}}{n}$$

注意，这些更新公式适用于归并 $n_i > 1$ 的两个数据分块。若任何分块只有单个数据（$n_i = 1$），则采用前一节中的增量更新公式。

这里有个示例阐明了对独立的统计概要所进行的汇集。此处要注意，任何 SummaryStatistics 的实例都可以进行序列化，并储存以供将来使用。

```
// 下列3个概要可以在不同时间从3台不同的机器上产生

SummaryStatistics ss1 = new SummaryStatistics();
ss1.addValue(1.0);
ss1.addValue(11.0);
ss1.addValue(5.0);

SummaryStatistics ss2 = new SummaryStatistics();
ss2.addValue(2.0);
ss2.addValue(12.0);
ss2.addValue(6.0);

SummaryStatistics ss3 = new SummaryStatistics();
ss3.addValue(0.0);
ss3.addValue(10.0);
ss3.addValue(4.0);

// 下列操作可以在上述操作完成之后的任意时间，在任意机器上进行

List<SummaryStatistics> ls = new ArrayList<>();
ls.add(ss1);
ls.add(ss2);
ls.add(ss3);

StatisticalSummaryValues s = AggregateSummaryStatistics.aggregate(ls);

System.out.println(s);
```

这将打印下面的报告，就像是对单一数据集进行计算那样：

```
StatisticalSummaryValues:
n: 9
min: 0.0
max: 12.0
mean: 5.666666666666667
std dev: 4.444097208657794
variance: 19.75
sum: 51.0
```

3.3.3 回归

因为矩与协矩可以容易地加在一起，所以 SimpleRegression 类使得回归变得容易。汇集统计产生与原始统计概要相同的结果。

```
SimpleRegression rg = new SimpleRegression();

/* Anscombe 4组数据中的x1与y1的x-y对 */
double[][] xyData = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
```

```
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

rg.addData(xyData);

/**/
double[][] xyData2 = {{10.0, 8.04}, {8.0, 6.95}, {13.0, 7.58},
        {9.0, 8.81}, {11.0, 8.33}, {14.0, 9.96}, {6.0, 7.24},
        {4.0, 4.26}, {12.0, 10.84}, {7.0, 4.82}, {5.0, 5.68}};

SimpleRegression rg2 = new SimpleRegression();
rg2.addData(xyData);

/* 对rg以及rg2进行回归的合并 */
rg.append(rg2);

/* 从组合的回归中获得回归结果 */
double alpha = rg.getIntercept(); // 3.0
double alpha_err = rg.getInterceptStdErr(); // 1.12
double beta = rg.getSlope(); // 0.5
double beta_err = rg.getSlopeStdErr(); // 0.12
double r2 = rg.getRSquare(); // 0.67
```

对于多元回归，`MillerUpdatingRegression` 类可通过 `MillerUpdatingRegression.addObservation(double[] x, double y)` 方法或者 `MillerUpdatingRegression.addObservations(double[][] x, double[] y)` 方法进行无存储的回归分析。

```
int numVars = 3;
boolean includeIntercept = true;
MillerUpdatingRegression r =
    new MillerUpdatingRegression(numVars, includeIntercept);
double[][] x = {{0, 0.5}, {1, 1.2}, {2, 2.5}, {3, 3.6}};
double[] y = {-1, 0.2, 0.9, 2.1};
r.addObservations(x, y);
RegressionResults rr = r.regress();
double[] params = rr.getParameterEstimates();
double[] errs = rr.getStdErrorOfEstimates();
double r2 = rr.getRSquared();
```

3.4 数据库内置函数的应用

大多数数据库有内置统计汇集函数。若数据已经在 MySQL 中，则不必把数据导入到 Java 应用中，而是可以使用内置函数。用 `GROUP BY` 以及 `ORDER BY` 并组合 `WHERE` 子句是一种将数据归约为统计概要的有效方式。记住，计算必须在某处进行，要么在应用中，要么在数据库服务器上。需要权衡的是，数据是否足够小，从而不会引起 CPU 与 I/O 的问题？若不想让数据库性能对 CPU 造成严重影响，则利用所有的 I/O 带宽就可以了。其他时候，宁可让 CPU 用于数据库应用，以计算所有统计值，只用一点点 I/O 开销把结果传回给等待的应用。



MySQL 中，内置函数 STDDEV 返回总体标准差，采用更具体的函数 STDDEV_SAMP 与 STDDEV_POP 可以分别返回样本标准差和总体标准差。

例如，可以用各种内置函数查询表格。以下是从销售量表格中得到的 AVG 与 STDDEV 收入统计的示例：

```
SELECT city, SUM(revenue) AS total_rev, AVG(revenue) AS avg_rev,
    STDDEV(revenue) AS std_rev
    FROM sales_table WHERE <some criteria> GROUP BY city ORDER BY total_rev DESC;
```

注意，可以直接使用从 JDBC 查询得到的结果，也可以把这些结果直接传入到构造方法 StatisticalSummaryValues(double mean, double variance, long count, double min, double max) 中，以供将来使用。假设有像下面这样的查询：

```
SELECT city, AVG(revenue) AS avg_rev,
             VAR_SAMP(revenue) AS var_rev,
             COUNT(revenue) AS count_rev,
MIN(revenue) AS min_rev, MAX(revenue) AS max_rev
    FROM sales_table WHERE <some criteria> GROUP BY city;
```

当通过数据库游标进行迭代时，可以（任意地）把每个 StatistialSummaryValues 实例添加到 List 或 Map 中，其键等于 city。

```
Map<String, StatisticalSummaryValues> revenueStats = new HashMap<>();

Statement st = c.createStatement();
ResultSet rs = st.executeQuery(selectSQL);
while(rs.next()) {
    StatisticalSummaryValues ss = new StatisticalSummaryValues(
        rs.getDouble("avg_rev"),
        rs.getDouble("var_rev"),
        rs.getLong("count_rev"),
        rs.getDouble("min_rev"),
        rs.getDouble("max_rev") );

    revenueStats.put(rs.getString("city"), ss);
}
rs.close();
st.close();
```

对于更大的数据集，一些简单而杰出的数据库操作，可以节省大量 I/O 开销。

第 4 章

数据操作

既然你已经知道如何把数据输入到有用的数据结构中，就可以利用所掌握的线性代数与统计学知识对数据进行操作。把数据提交给学习算法之前，可以对数据进行多种操作，这通常称为**预处理**（preprocessing）。预处理包括数据清理、对数据归一化或缩放、把数据归约至较小的规模、把文本值编码为数字值，以及把数据拆分为不同部分以用于训练模型与测试模型。通常，数据已经具有一种或者另一种格式（例如 `List` 或 `double[][]`），学习例程可能采用任何一种格式，或者两种格式都用。此外，学习算法可能需要知道标签是二值的还是多类别的，或者是采用其他方式编码的，例如文本。上述这些都需要考虑，并且要在数据进入学习算法之前做好准备。对于从源头得到原始数据，然后将其准备好用于学习算法或预测算法的这一自动化工作流程，本章的步骤可以成为其中的一部分。

4.1　转换文本数据

许多学习算法或预测算法都需要数值输入。完成这一操作最简单的方式之一是创建向量空间模型，在其中定义一个已知维数的向量，然后把文本片段（乃至单词）的集合赋值给对应的向量集合。把文本转换为向量的一般过程有多种选择和变体。此处假定存在大量文本（语料库），可以把它们划分为句子或行（文档），然后进一步分为单词（标记）。注意**语料库**（corpus）、**文档**（document）以及标记（token）由用户定义。

4.1.1　从文档中提取标记

对于每个文档，需要提取所有标记。因为有许多方法来处理这个问题，所以可以创建一个接口。该接口有个方法，该方法以文档字符串为输入，返回由 `String` 标记组成的数组。

```
public interface Tokenizer {
    String[] getTokens(String document);
}
```

标记可能有许多不需要的字符，例如标点符号、数字或者其他字符。当然，这完全依赖于应用。在本例中，只关心常规英语单词的实际内容，因此可以清理标记，使其只接受小写字母字符。引入最小标记大小的变量，可以略过诸如 a、or、at 等单词。

```
public class SimpleTokenizer implements Tokenizer {

    private final int minTokenSize;

    public SimpleTokenizer(int minTokenSize) {
        this.minTokenSize = minTokenSize;
    }

    public SimpleTokenizer() {
        this(0);
    }

    @Override
    public String[] getTokens(String document) {
        String[] tokens = document.trim().split("\\s+");
        List<String> cleanTokens = new ArrayList<>();
        for (String token : tokens) {
            String cleanToken = token.trim().toLowerCase()
                .replaceAll("[^A-Za-z\']+", "");
            if(cleanToken.length() > minTokenSize) {
                cleanTokens.add(cleanToken);
            }
        }
        return cleanTokens.toArray(new String[0]);
    }
}
```

4.1.2 利用字典

字典（dictionary）是相关词语的列表（即“词汇表”）。实现字典的策略不止一种。重要的特征是，每个词语需要与对应于其在向量中位置的整数值相联系。当然，这可以是能通过位置查找的数组，但是对于大字典，数组是低效的，采用 `Map` 更好。对于非常大的字典，可以略过词语存储，采用散列方法。通常来说，需要知道字典中词语的数量，以便于创建向量，还要知道返回特定词语索引的方法。注意整型（`int`）不能是 `null`，因此采用装箱类型 `Integer`，可以使得返回索引是 `int` 或 `null`。

```
public interface Dictionary {
    Integer getTermIndex(String term);
    int getNumTerms();
}
```

可以构造从 Tokenizer 实例中搜集得到的准确词语的字典。注意这里的策略是为每个项添加一个词语以及一个整数。新项将使计数器递增；重复项将被丢弃，而不使计数器递增。在这种情况下，TermDictionary 类需要添加新项的方法。

```
public class TermDictionary implements Dictionary {

    private final Map<String, Integer> indexedTerms;
    private int counter;

    public TermDictionary() {
        indexedTerms = new HashMap<>();
        counter = 0;
    }

    public void addTerm(String term) {
        if(!indexedTerms.containsKey(term)) {
            indexedTerms.put(term, counter++);
        }
    }

    public void addTerms(String[] terms) {
        for (String term : terms) {
            addTerm(term);
        }
    }

    @Override
    public Integer getTermIndex(String term) {
        return indexedTerms.get(term);
    }

    @Override
    public int getNumTerms() {
        return indexedTerms.size();
    }
}
```

对于大量词语，可以采用散列方法。基本上，用每个词语 String 值的散列码，然后用该散列码取字典中词语总数的模。当词语总数很大时（大约 100 万），不太可能发生冲突。注意，与 TermDictionary 不同，不需要添加词语或记录词语。在运行时将计算每个词语的索引。词语总数是所设置的常量。为了有效地访问散列表，把词语总数设置为 2^n 是个不错的主意。设置为大约 2^{20} 时，大约是 100 万个词语。

```
public class HashingDictionary implements Dictionary {

    private int numTerms; // 2^n是最优值

    public HashingDictionary() {
        // 2^20 = 1048576
        this(new Double(Math.pow(2,20)).intValue());
    }
```

```
    public HashingDictionary(int numTerms) {
        this.numTerms = numTerms;
    }

    @Override
    public Integer getTermIndex(String term) {
        return Math.floorMod(term.hashCode(), numTerms);
    }

    @Override
    public int getNumTerms() {
        return numTerms;
    }
}
```

4.1.3 文档向量化

有了分词器（tokenizer）以及字典之后，就可以把单词列表转换为能传递给机器学习算法的数值。最直接的方法是首先确定字典是什么，然后统计在句子（或需要关注的文本）中出现的次数。通常把这种方法称为**词袋**（bag of words）。有些时候，只想知道单词是否出现了。此时，在向量中放置 1，而不是记录单词出现的次数。

```
public class Vectorizer {

    private final Dictionary dictionary;
    private final Tokenizer tokenzier;
    private final boolean isBinary;

    public Vectorizer(Dictionary dictionary, Tokenizer tokenzier,
        boolean isBinary) {
        this.dictionary = dictionary;
        this.tokenzier = tokenzier;
        this.isBinary = isBinary;
    }

    public Vectorizer() {
        this(new HashingDictionary(), new SimpleTokenizer(), false);
    }

    public RealVector getCountVector(String document) {
        RealVector vector = new OpenMapRealVector(dictionary.getNumTerms());
        String[] tokens = tokenzier.getTokens(document);
        for (String token : tokens) {
            Integer index = dictionary.getTermIndex(token);
            if(index != null) {
                if(isBinary) {
                    vector.setEntry(index, 1);
                } else {
                    vector.addToEntry(index, 1); // 增量!
                }
            }
```

```
        }
        return vector;
    }

    public RealMatrix getCountMatrix(List<String> documents) {
        int rowDimension = documents.size();
        int columnDimension = dictionary.getNumTerms();
        RealMatrix matrix = new OpenMapRealMatrix(rowDimension, columnDimension);
        int counter = 0;
        for (String document : documents) {
            matrix.setRowVector(counter++, getCountVector(document));
        }
        return matrix;
    }
}
```

有时候，我们想要减少常用词的作用。词频－逆向文档频率（TFIDF）向量正是做这个事的。当词语在少数文档中出现许多次时，TFIDF 分量最高；当词语几乎出现在所有文档中时，TFIDF 分量最低。注意，TFIDF 正是词频乘以逆向文档频率：TFIDF = TF × IDF。其中 TF 是词语在文档中出现的次数（词语的计数向量），DF 是包含该词语的文档个数，IDF 则是文档频率（DF）的（伪）倒数。通常，可以逐个统计文档的词语来计算 TF。对每个文档计算二进制向量，随后处理每个文档时，对这些向量累计求和以得到 DF。TFIDF 最常见的形式如下，其中 N 是所处理文档的总数。

$$\mathrm{TFIDF}_{t,d} = \mathrm{TF}_{t,d} \log(N / \mathrm{DF}_t)$$

这仅是计算 TFIDF 的策略之一。注意，若 N 或 DF 值等于 0，对数函数会出现问题。某些策略通过加上小因子或 1，以避免这个问题。在实现时，可以通过设置 log(0) 等于 0 来应对这个问题。通常，在实现时，首先创建关于词语计数的矩阵，然后对该矩阵进行操作，把每个词语转换为其加权的 TFIDF 值。因为这些矩阵通常是稀疏的，所以采用优化的按照数量级进行游走的算子是个好主意。

```
public class TFIDF implements RealMatrixChangingVisitor {

    private final int numDocuments;
    private final RealVector termDocumentFrequency;
    double logNumDocuments;

    public TFIDF(int numDocuments, RealVector termDocumentFrequency) {
        this.numDocuments = numDocuments;
        this.termDocumentFrequency = termDocumentFrequency;
        this.logNumDocuments = numDocuments > 0 ? Math.log(numDocuments) : 0;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
            int startColumn, int endColumn) {
        // 不匹配
```

```
    }

    @Override
    public double visit(int row, int column, double value) {
        double df = termDocumentFrequency.getEntry(column);
        double logDF = df > 0 ? Math.log(df) : 0.0;
        // TFIDF = TF_i * log(N/DF_i) = TF_i * ( log(N) - log(DF_i) )
        return value * (logNumDocuments - logDF);
    }

    @Override
    public double end() {
        return 0.0;
    }

}
```

于是，TFIDFVectorizer 用到了（针对词语的）计数以及二进制计数。

```
public class TFIDFVectorizer {

    private Vectorizer vectorizer;
    private Vectorizer binaryVectorizer;
    private int numTerms;

    public TFIDFVectorizer(Dictionary dictionary, Tokenizer tokenzier) {
        vectorizer = new Vectorizer(dictionary, tokenzier, false);
        binaryVectorizer = new Vectorizer(dictionary, tokenzier, true);
        numTerms = dictionary.getNumTerms();
    }

    public TFIDFVectorizer() {
        this(new HashingDictionary(), new SimpleTokenizer());
    }

    public RealVector getTermDocumentCount(List<String> documents) {
        RealVector vector = new OpenMapRealVector(numTerms);
        for (String document : documents) {
            vector.add(binaryVectorizer.getCountVector(document));
        }
        return vector;
    }

    public RealMatrix getTFIDF(List<String> documents) {
        int numDocuments = documents.size();
        RealVector df = getTermDocumentCount(documents);
        RealMatrix tfidf = vectorizer.getCountMatrix(documents);
        tfidf.walkInOptimizedOrder(new TFIDF(numDocuments, df));
        return tfidf;
    }
}
```

下面是附录 A 给出的观点数据集（sentiment dataset）的应用示例。

```
/* 观点数据……见附录A */
Sentiment sentiment = new Sentiment();

/* 创建所有词语的字典 */
TermDictionary termDictionary = new TermDictionary();

/* 需要基本的分词器来对文本进行解析 */
SimpleTokenizer tokenizer = new SimpleTokenizer();

/* 把观点数据集中的所有词语添加到字典中 */
for (String document : sentiment.getDocuments()) {
    String[]tokens = tokenizer.getTokens(document);
    termDictionary.addTerms(tokens);
}

/* 创建关于每个句子单词数的矩阵 */
Vectorizer vectorizer = new Vectorizer(termDictionary, tokenizer, false);
RealMatrix counts = vectorizer.getCountMatrix(sentiment.getDocuments());

/* ……或者创建二进制计数器 */
Vectorizer binaryVectorizer = new Vectorizer(termDictionary, tokenizer, true);
RealMatrix binCounts = binaryVectorizer.getCountMatrix(sentiment.getDocuments());

/* ……或者创建TFIDF矩阵 */
TFIDFVectorizer tfidfVectorizer = new TFIDFVectorizer(termDictionary, tokenizer);
RealMatrix tfidf = tfidfVectorizer.getTFIDF(sentiment.getDocuments());
```

4.2 数值数据的缩放与归一化

应当从类中获得数据还是使用数组？此处的目标是对数据集中的每个元素进行某种变换，即 $f(x_{i,j}) \rightarrow x_{i,j}^*$。对数据进行缩放有两种基本方式：按列或按行。按列缩放时，只需要收集每一列数据的统计量。具体说来，需要最小值、最大值、均值以及标准差。因此，如果把整个数据集添加到 `MultivariateSummaryStatistics` 的实例中，将得到上述所有统计量。按行缩放时，需要对每一行进行 L1 或 L2 归一化。可以把这些值存储在 `RealVector` 的实例中，它可能是稀疏的。



若对数据进行缩放以训练模型，则要保留所使用过的任何最小值、最大值、均值或标准差。在转换将用来做预测的新数据集时，必须使用相同的技术，包括所存储的参数。注意，如果把数据拆分为训练集、验证集和测试集，那么要对训练集数据进行缩放，并用那些值（例如均值）对验证集和测试集进行缩放，使得缩放后的数据集是无偏的。

4.2.1 对列进行缩放

对列进行缩放的一般形式是采用 `RealMatrixChangingVisitor` 类，将预先算好的关于列的统计量传入构造方法中。当对矩阵的每个元素进行运算时，会用到相应的列统计量。

```
public class MatrixScalingOperator implements RealMatrixChangingVisitor {

    MultivariateSummaryStatistics mss;

    public MatrixScalingOperator(MultivariateSummaryStatistics mss) {
        this.mss = mss;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
        int startColumn, int endColumn) {
        // 什么也不做
    }

    @Override
    public double visit(int row, int column, double value) {
        // 此处实现特定的类型
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

1. 最小值 – 最大值缩放

最小值 − 最大值缩放可以确保每列各自的最小值为 0，最大值为 1。可以对第 j 列的每个元素 i 用该列的最小值与最大值进行变换。

$$x^{*}_{i,j} = \frac{x_{i,j} - x_j^{\min}}{x_j^{\max} - x_j^{\min}}$$

这可以通过下面的代码实现：

```
public class MatrixScalingMinMaxOperator implements RealMatrixChangingVisitor {
...
    @Override
    public double visit(int row, int column, double value) {
        double min = mss.getMin()[column];
        double max = mss.getMax()[column];
        return (value - min) / (max - min);
    }
...
}
```

有时候想要设定下限 a 与上限 b（而不是 0 与 1）。此时，先计算关于下限为 0、上限为 1 的数据缩放，再进行第二轮缩放。

$$x^{a,b}_{i,j} = x^{*}_{i,j}(b-a)+a$$

2. 将数据居中

将数据相对于均值居中，可使得列数据的平均值变为 0。然而，因为所处理的数据是没有大小限制的，所以仍然有异常的最小值与最大值。一列中的每个值可以根据该列的均值进行变换。

$$x_{i,j}^{*} = x_{i,j} - \overline{x}_j$$

实现上述操作的代码如下：

```
@Override
public double visit(int row, int column, double value) {
    double mean = mss.getMean()[column];
    return value - mean;
}
```

3. 单位正态缩放

单位正态缩放也称作 ***z* 分数**（z-score）。它对列中的每个数据点进行两次缩放，先以均值为中心使数据居中，再除以标准差，使得数据点成为单位正态分布的一员。变换之后，每一列的平均值将是 0，该分布中大多数值会小于 1，虽然这是一种分布，但无法保证这一点，因为数值大小是不受限制的。

$$x_{i,j}^{*} = \frac{x_{i,j} - \overline{x}_j}{\sigma_j}$$

实现代码如下：

```
@Override
public double visit(int row, int column, double value) {
    double mean = mss.getMean()[column];
    double std = mss.getStandardDeviation()[column];
    return (value - mean) / std;
}
```

4.2.2　对行进行缩放

当每行数据都是关于所有变量的一条记录时，对行数据缩放通常是进行 L1 或者 L2 归一化。

```
public class MatrixScalingOperator implements RealMatrixChangingVisitor {

    RealVector normals;

    public MatrixScalingOperator(RealVector normals) {
        this.normals = normals;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
```

```
            int startColumn, int endColumn) {
            // 什么也不做
        }

        @Override
        public double visit(int row, int column, double value) {
            // 实现
        }

        @Override
        public double end() {
            return 0.0;
        }
    }
```

1. L1 归一化

在这种情况下，对每行数据进行归一化，使得每一行（绝对）值之和为 1，方法是把第 i 行的每个元素 j 都除以该行的 L1 范数。

$$x_{i,j}^{*} = \frac{x_{i,j}}{|\boldsymbol{x}_i|}$$

实现代码如下：

```
    @Override
    public double visit(int row, int column, double value) {
        double rowNormal = normals.getEntry(row);
        return ( rowNormal > 0 ) ? value / rowNormal : 0;
    }
```

2. L2 归一化

同样，L2 归一化也是针对行缩放，而不是针对列。在这种情况下，对第 i 行的每个元素 j 除以该行的 L2 范数来对每一行数据进行归一化。之后，每一行的长度等于 1。

$$x_{i,j}^{*} = \frac{x_{i,j}}{\|\boldsymbol{x}_i\|}$$

实现代码如下：

```
    @Override
    public double visit(int row, int column, double value) {
        double rowNormal = normals.getEntry(row);
        return ( rowNormal > 0 ) ? value / rowNormal : 0;
    }
```

4.2.3　矩阵的缩放算子

因为要就地修改矩阵，所以可以收集缩放算法，采用静态方法来编写。

```
public class MatrixScaler {

    public static void minmax(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(new MatrixScalingMinMaxOperator(mss));
    }

    public static void center(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(
            new MatrixScalingOperator(mss, MatrixScaleType.CENTER));
    }

    public static void zscore(RealMatrix matrix) {
        MultivariateSummaryStatistics mss = getStats(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(mss, MatrixScaleType.ZSCORE));
    }

    public static void l1(RealMatrix matrix) {
        RealVector normals = getL1Normals(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(normals, MatrixScaleType.L1));
    }

    public static void l2(RealMatrix matrix) {
        RealVector normals = getL2Normals(matrix);
        matrix.walkInOptimizedOrder(
          new MatrixScalingOperator(normals, MatrixScaleType.L2));
    }

    private static RealVector getL1Normals(RealMatrix matrix) {
        RealVector normals = new OpenMapRealVector(matrix.getRowDimension());
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            double l1Norm = matrix.getRowVector(i).getL1Norm();
            if (l1Norm > 0) {
                normals.setEntry(i, l1Norm);
            }
        }
        return normals;
    }

    private static RealVector getL2Normals(RealMatrix matrix) {
        RealVector normals = new OpenMapRealVector(matrix.getRowDimension());
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            double l2Norm = matrix.getRowVector(i).getNorm();
            if (l2Norm > 0) {
                normals.setEntry(i, l2Norm);
            }
        }
        return normals;
    }

    private static MultivariateSummaryStatistics getStats(RealMatrix matrix) {
        MultivariateSummaryStatistics mss =
```

```
        new MultivariateSummaryStatistics(matrix.getColumnDimension(), true);
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            mss.addValue(matrix.getRow(i));
        }
        return mss;
    }
}
```

现在就很容易使用缩放算法了。

```
RealMatrix matrix = new OpenMapRealMatrix(10, 3);
        matrix.addToEntry(0, 0, 1.0);
        matrix.addToEntry(0, 2, 2.0);
        matrix.addToEntry(1, 0, 1.0);
        matrix.addToEntry(2, 0, 3.0);
        matrix.addToEntry(3, 1, 5.0);
        matrix.addToEntry(6, 2, 1.0);
        matrix.addToEntry(8, 0, 8.0);
        matrix.addToEntry(9, 1, 3.0);

/* 就地对矩阵进行缩放 */
MatrixScaler.minmax(matrix);
```

4.3 将数据降维至主成分

主成分分析（principal components analysis，PCA）的目标是把某个数据集变换为另一个维数较少的数据集。可以把其看作对 $m \times n$ 矩阵 $\boldsymbol{X}$ 应用函数 f，结果是 $m \times k$ 矩阵 $\boldsymbol{X}_k$，即 $\boldsymbol{X}_k = f(\boldsymbol{X})$，其中 $k < n$。

通过线性代数算法寻找特征向量与特征值，可以完成该操作。这种变换的好处之一是，新的维度按照从最主要的到最次要的进行排列。对于多维数据，有时可以通过绘制最主要的两个维度的数据来了解任何重要的关系。在图 4-1 中，绘制了鸢尾属植物（Iris）数据集最主要的两个成分（见附录 A）。Iris 数据集是四维特征集，有 3 个可能的标签。在这个图中，通过绘制投影到两个最主要成分的原始数据，可以看到 3 种类别的区别。若从原始数据集绘制任意两个维度，则并不能看出这种区别。

然而，对于高维数据，需要一种更稳健的方式来决定需要保留的主成分个数。因为主成分是按照从最主要的到最次要的进行排列的，所以可通过计算归一化的关于特征值 λ 的累加和，来确切表示主成分的可释方差（explained variance）。

$$\sigma^2(k) = \frac{1}{\sigma^2_{\max}} \sum_{i=1}^{k} \lambda_i$$

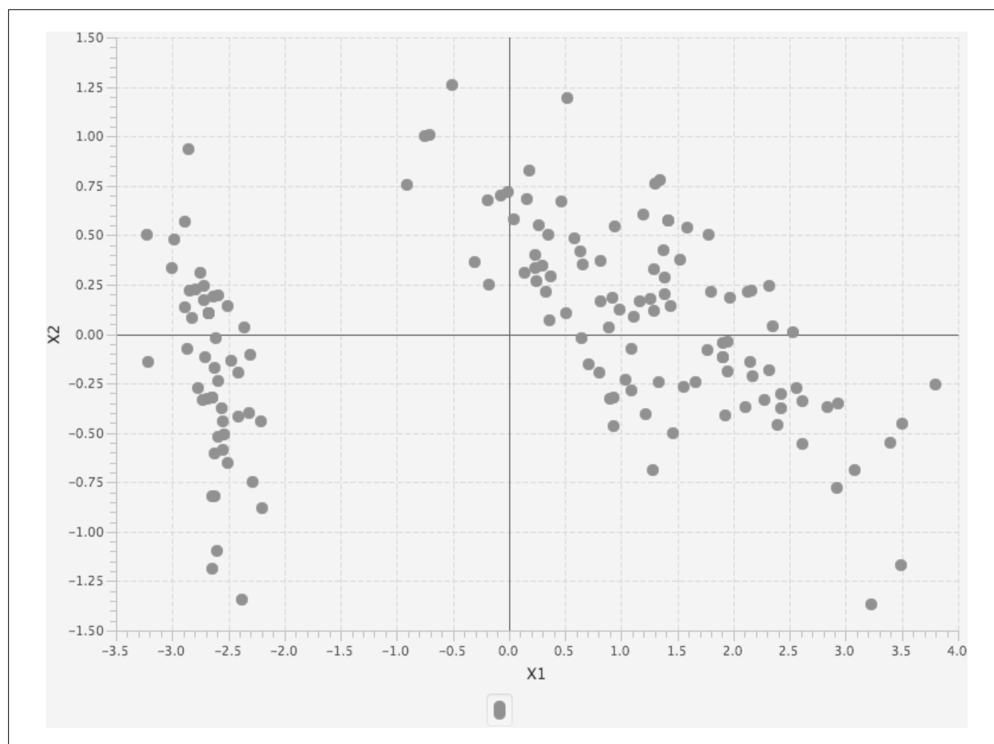


图 4-1：Iris 数据集的前两个主成分

此处，每个额外的成分解释了额外百分比的数据。于是，可释方差有两种用处。当显式地选择一定数目的主成分时，可以计算出多少原始数据可以被这种新的变换所解释。另一种情况下，可以对可释方差向量进行迭代，当有特定数目（例如 *k* 个）主成分能够解释期望比例的原始数据时，就可以停止迭代。

实现主成分分析时，有几种计算特征值与特征向量的策略。最终，仅需要获取变换后的数据。在 PCA 主类中仅给出框架，而在另外的类中给出实现细节，是一种不错的策略模式。

```
public class PCA {

    private final PCAImplementation pCAImplementation;

    public PCA(RealMatrix data, PCAImplementation pCAImplementation) {
        this.pCAImplementation = pCAImplementation;
        this.pCAImplementation.compute(data);
    }

    public RealMatrix getPrincipalComponents(int k) {
        return pCAImplementation.getPrincipalComponents(k);
    }
```

```
    public RealMatrix getPrincipalComponents(int k, RealMatrix otherData) {
        return pCAImplementation.getPrincipalComponents(k, otherData);
    }

    public RealVector getExplainedVariances() {
        return pCAImplementation.getExplainedVariances();
    }

    public RealVector getCumulativeVariances() {
        RealVector variances = getExplainedVariances();
        RealVector cumulative = variances.copy();
        double sum = 0;
        for (int i = 0; i < cumulative.getDimension(); i++) {
            sum += cumulative.getEntry(i);
            cumulative.setEntry(i, sum);
        }
        return cumulative;
    }

    public int getNumberOfComponents(double threshold) {
        RealVector cumulative = getCumulativeVariances();
        int numComponents=1;
        for (int i = 0; i < cumulative.getDimension(); i++) {
            numComponents = i + 1;
            if(cumulative.getEntry(i) >= threshold) {
                break;
            }
        }
        return numComponents;
    }

    public RealMatrix getPrincipalComponents(double threshold) {
        int numComponents = getNumberOfComponents(threshold);
        return getPrincipalComponents(numComponents);
    }

    public RealMatrix getPrincipalComponents(double threshold,
        RealMatrix otherData) {
        int numComponents = getNumberOfComponents(threshold);
        return getPrincipalComponents(numComponents, otherData);
    }

}
```

然后，可以为把输入数据分解为其主成分的后续方法提供一个 PCAImplementation 接口。

```
public interface PCAImplementation {

    void compute(RealMatrix data);

    RealVector getExplainedVariances();

    RealMatrix getPrincipalComponents(int numComponents);

    RealMatrix getPrincipalComponents(int numComponents, RealMatrix otherData);
}
```

4.3.1 协方差方法

计算 PCA 的一种方法是寻找 $\boldsymbol{X}$ 的协方差矩阵的特征分解。（列数据关于列均值）居中的矩阵 $\boldsymbol{X}$ 的主成分是协方差矩阵的特征向量。

$$\boldsymbol{C} = \frac{1}{n-1}\left(\boldsymbol{X} - \bar{\boldsymbol{X}}\right)^{\mathrm{T}}\left(\boldsymbol{X} - \bar{\boldsymbol{X}}\right)$$

因为需要将两个可能很大的矩阵相乘，所以这种协方差计算方法是计算密集的。然而，第 3 章中探讨了关于计算协方差的有效更新公式，它不需要矩阵转置。当使用 Apache Commons Math 的 `Covariance` 类，或者其他实现了它的类（例如 `MultivariateSummaryStatistics` 类）时，采用了有效的更新公式。因此，协方差矩阵 $\boldsymbol{C}$ 可以分解为下式：

$$\boldsymbol{C} = \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{\mathrm{T}}$$

其中，$\boldsymbol{V}$ 的列是特征向量，$\boldsymbol{D}$ 的对角元素是特征值。Apache Commons Math 的实现把特征值（以及相应的特征向量）从最大到最小排列。通常，只需要 k 个成分，因此只需要 $\boldsymbol{V}$ 的前 k 列。通过矩阵相乘，可以把相对于均值居中的数据投影到新成分上。

$$\boldsymbol{X}_k = \left(\boldsymbol{X} - \bar{\boldsymbol{X}}\right)\boldsymbol{V}_k$$

以下是采用协方差方法实现的主成分分析：

```
public class PCAEIGImplementation implements PCAImplementation {

    private RealMatrix data;
    private RealMatrix d; // 特征值矩阵
    private RealMatrix v; // 特征向量矩阵
    private RealVector explainedVariances;
    private EigenDecomposition eig;
    private final MatrixScaler matrixScaler;

    public PCAEIGImplementation() {
        matrixScaler = new MatrixScaler(MatrixScaleType.CENTER);
    }

    @Override
    public void compute(RealMatrix data) {
        this.data = data;
        eig = new EigenDecomposition(new Covariance(data).getCovarianceMatrix());
        d = eig.getD();
        v = eig.getV();
    }

    @Override
    public RealVector getExplainedVariances() {
        int n = eig.getD().getColumnDimension(); //colD = rowD
        explainedVariances = new ArrayRealVector(n);
```

```
        double[] eigenValues = eig.getRealEigenvalues();
        double cumulative = 0.0;
        for (int i = 0; i < n; i++) {
            double var = eigenValues[i];
            cumulative += var;
            explainedVariances.setEntry(i, var);
        }
        /* 用向量除以最终（最大）的累加和，使得可释方差最大为1 */
        return explainedVariances.mapDivideToSelf(cumulative);
    }

    @Override
    public RealMatrix getPrincipalComponents(int k) {
        int m = eig.getV().getColumnDimension(); // rowD = colD
        matrixScaler.transform(data);
        return data.multiply(eig.getV().getSubMatrix(0, m-1, 0, k-1));
    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents,
        RealMatrix otherData) {
        int numRows = v.getRowDimension();
        matrixScaler.transform(otherData);
        return otherData.multiply(
            v.getSubMatrix(0, numRows-1, 0, numComponents-1));
    }
}
```

例如，接下来可以用 PCAEIGImplementation 获得前 3 个主成分，或者获得可以提供 50% 可释方差的所有成分：

```
/* 采用特征分解实现 */
PCA pca = new PCA(data, new PCAEIGImplementation());

/* 获得前3个成分 */
RealMatrix pc3 = pca.getPrincipalComponents(3);

/* 获得能够满足50%可释方差所需的那么多成分 */
RealMatrix pct = pca.getPrincipalComponents(.5);
```

4.3.2　SVD方法

若$\boldsymbol{X}-\bar{\boldsymbol{X}}$是包含 m 行 n 列且相对于均值居中的数据集，则主成分可以按下式计算：

$$\boldsymbol{X}-\bar{\boldsymbol{X}}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}}$$

注意，对于我们熟悉的奇异值分解$\boldsymbol{A}=\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{T}}$，矩阵 $\boldsymbol{V}$ 的列向量就是特征向量，特征值可以从矩阵 $\boldsymbol{\Sigma}$ 的对角线通过$\lambda_i=\sum_{i,i}^2/(m-1)$得到，$m$ 是数据行数。完成了相对于均值居中的矩阵 $\boldsymbol{X}$ 的奇异值分解之后，投影如下式：

$$\boldsymbol{X}_k=\boldsymbol{U}_k\boldsymbol{\Sigma}_k$$

此处只保留了矩阵 $\boldsymbol{U}$ 的前 k 列以及矩阵 $\boldsymbol{\Sigma}$ 左上角的 $k \times k$ 子阵。采用原先相对于均值居中的数据以及特征向量计算投影也是正确的。

$$\boldsymbol{X}_k = \left(\boldsymbol{X} - \bar{\boldsymbol{X}}\right)\boldsymbol{V}_k$$

此处只保留了矩阵 $\boldsymbol{V}$ 的前 k 列。当采用已有特征向量对新数据集进行转换时，采用这个表达式。这个形式与前一节的特征分解方法相同。

因为最多有 $p = \min(m, n)$ 个奇异值，所以 Apache Commons Math 实现的是**瘦**（compact）SVD，于是，正如第 2 章讨论的那样，不需要计算完全的 SVD。下面是主成分分析的一种 SVD 实现，也是推荐的方法。

```
public class PCASVDImplementation implements PCAImplementation {

    private RealMatrix u;
    private RealMatrix s;
    private RealMatrix v;
    private MatrixScaler matrixScaler;
    private SingularValueDecomposition svd;

    @Override
    public void compute(RealMatrix data) {
        MatrixScaler.center(data);
        svd = new SingularValueDecomposition(data);
        u = svd.getU();
        s = svd.getS();
        v = svd.getV();
    }

    @Override
    public RealVector getExplainedVariances() {
        double[] singularValues = svd.getSingularValues();
        int n = singularValues.length;
        int m = u.getRowDimension(); // U的行数与数据中的相同
        RealVector explainedVariances = new ArrayRealVector(n);
        double sum = 0.0;
        for (int i = 0; i < n; i++) {
            double var = Math.pow(singularValues[i], 2) / (double)(m-1);
            sum += var;
            explainedVariances.setEntry(i, var);
        }
        /* 用向量除以最终（最大）的累加和，使得可解释方差最大为1*/
        return explainedVariances.mapDivideToSelf(sum);

    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents) {
        int numRows = svd.getU().getRowDimension();
        /* 子阵的边界是包含在内的 */
        RealMatrix uk = u.getSubMatrix(0, numRows-1, 0, numComponents-1);
```

```
        RealMatrix sk = s.getSubMatrix(0, numComponents-1, 0, numComponents-1);
        return uk.multiply(sk);
    }

    @Override
    public RealMatrix getPrincipalComponents(int numComponents,
        RealMatrix otherData) {
        matrixScaler.transform(otherData);
        int numRows = v.getRowDimension();
        // 子阵的索引是包含在内的
        return otherData.multiply(v.getSubMatrix(0, numRows-1, 0, numComponents-1));
    }
}
```

然后，为了实现它，采用下列代码：

```
/* 采用奇异值分解实现  */
PCA pca = new PCA(data, new PCASVDImplementation());

/* 获得前3个成分 */
RealMatrix pc3 = pca.getPrincipalComponents(3);

/* 获得能够满足50%可释方差所需的那么多成分 */
RealMatrix pct = pca.getPrincipalComponents(.5);
```

4.4 创建训练集、验证集及测试集

对于监督型学习，用数据集的一部分建立模型，然后用测试集进行预测，（用测试集的已知标签）看预测是否正确。有时，在训练过程中需要第三个集合，以验证模型参数，这个集合称作**验证集**（validation set）。

用训练集训练模型，用验证集选择模型，最后用测试集计算模型误差。我们至少有两种选择。第一种是从随机整数中采样，然后根据这些采样的随机整数选择数组或矩阵的一部分采样。第二种是对数据自身重新打乱次序，成为 `List`，并提取一些子列表。在所提取的这些子列表中，每种类型的数据集长度必须符合需要。

4.4.1 基于索引的重新采样

为数据集的每个点创建索引。

```
public class Resampler {

    RealMatrix features;
    RealMatrix labels;
    List<Integer> indices;
    List<Integer> trainingIndices;
    List<Integer> validationIndices;
    List<Integer> testingIndices;
```

```
    int[] rowIndices;
    int[] test;
    int[] validate;

    public Resampler(RealMatrix features, RealMatrix labels) {
        this.features = features;
        this.labels = labels;
        indices = new ArrayList<>();
    }

    public void calculateTestTrainSplit(double testFraction, long seed) {
        Random rnd = new Random(seed);
        for (int i = 1; i <= features.getRowDimension(); i++) {
            indices.add(i);
        }
        Collections.shuffle(indices, rnd);
        int testSize = new Long(Math.round(
        testFraction * features.getRowDimension())).intValue();
        /* subList包括起始索引，但是不包括结束索引 */
        testingIndices = indices.subList(0, testSize);
        trainingIndices = indices.subList(testSize, features.getRowDimension());
    }

    public RealMatrix getTrainingFeatures() {
        int numRows = trainingIndices.size();
        rowIndices = new int[numRows];
        int counter = 0;
        for (Integer trainingIndex : trainingIndices) {
            rowIndices[counter] = trainingIndex;
        }
        counter++;
        int numCols = features.getColumnDimension();
        int[] columnIndices = new int[numCols];
        for (int i = 0; i < numCols; i++) {
            columnIndices[i] = i;
        }
        return features.getSubMatrix(rowIndices, columnIndices);
    }
}
```

此处是应用 Iris 数据集的示例。

```
Iris iris = new Iris();

Resampler resampler = new Resampler(iris.getFeatures(), iris.getLabels());
resampler.calculateTestTrainSplit(0.40, 0L);

RealMatrix trainFeatures = resampler.getTrainingFeatures();
RealMatrix trainLabels = resampler.getTrainingLabels();
RealMatrix testFeatures = resampler.getTestingFeatures();
RealMatrix testLabels = resampler.getTestingLabels();
```

4.4.2 基于列表的重新采样

在一些情况下，可能已经把数据定义为对象的集合。例如，可能有关于 `Record` 类型的 `List`，其中包含每个数据记录（行）的数据。于是，直接的方法是建立基于 `List` 的重新采样器，它采用泛型 `T`。

```
public class Resampling<T> {

    private final List<T> data;
    private final int trainingSetSize;
    private final int testSetSize;
    private final int validationSetSize;

    public Resampling(List<T> data, double testFraction, long seed) {
        this(data, testFraction, 0.0, seed);
    }

    public Resampling(List<T> data, double testFraction,
    double validationFraction, long seed) {
        this.data = data;
        validationSetSize = new Double(
            validationFraction * data.size()).intValue();
        testSetSize = new Double(testFraction * data.size()).intValue();
        trainingSetSize = data.size() - (testSetSize + validationSetSize);
        Random rnd = new Random(seed);
        Collections.shuffle(data, rnd);
    }

    public int getTestSetSize() {
        return testSetSize;
    }

    public int getTrainingSetSize() {
        return trainingSetSize;
    }

    public int getValidationSetSize() {
        return validationSetSize;
    }

    public List<T> getValidationSet() {
        return data.subList(0, validationSetSize);
    }

    public List<T> getTestSet() {
        return data.subList(validationSetSize, validationSetSize + testSetSize);
    }

    public List<T> getTrainingSet() {
        return data.subList(validationSetSize + testSetSize, data.size());
    }
}
```

给定预先定义的 Record 类，可以像这样使用重新采样器：

```
Resampling<Record> resampling = new Resampling<>(data, 0.20, 0L);
// Resampling<Record> resampling = new Resampling<>(data, 0.20, 0.20, 0L);
List<Record> testSet = resampling.getTestSet();
List<Record> trainingSet = resampling.getTrainingSet();
List<Record> validationSet = resampling.getValidationSet();
```

4.4.3 小批量

在几种学习算法中，从大得多的数据集中随机采样（100 个数据点的规模）小批量输入数据是有利的。在这个任务中，可以重用 MatrixResampler 的代码。重要的是要记住，当指定批量大小时，要特别指明是测试集，而不是训练集，如同 MatrixResampler 中实现的那样。

```
public class Batch extends MatrixResampler {

    public Batch(RealMatrix features, RealMatrix labels) {
        super(features, labels);
    }

    public void calcNextBatch(int batchSize) {
        super.calculateTestTrainSplit(batchSize);
    }

    public RealMatrix getInputBatch() {
        return super.getTestingFeatures();
    }

    public RealMatrix getTargetBatch() {
        return super.getTestingLabels();
    }
}
```

4.5 标签的编码

若拿到的标签是文本字段，例如 red 或 blue，则可以把它们转换为整数，以供进一步处理。



处理分类算法时，把输出变量每个独特的实例称作**类别**（class）。回想一下，class 是 Java 的关键字，因此我们必须用别的术语，例如 className、classLabel 或者 classes（对于复数）。当用 classes 作为 List 的名称来编写 for...each 循环时，要注意集成开发环境（IDE）的代码自动补全功能。

4.5.1 泛型编码器

以下是关于泛型 T 的标签编码器的实现，注意它创建从 0 到 $n - 1$ 的类别。换句话说，结果类别就是其在 ArrayList 中的位置。

```java
public class LabelEncoder<T> {

    private final List<T> classes;

    public LabelEncoder(T[] labels) {
        classes = Arrays.asList(labels);
    }

    public List<T> getClasses() {
        return classes;
    }

    public int encode(T label) {
        return classes.indexOf(label);
    }

    public T decode(int index) {
        return classes.get(index);
    }
}
```

下面是采用真实数据编码标签的示例：

```java
String[] stringLabels = {"Sunday", "Monday", "Tuesday"};

LabelEncoder<String> stringEncoder = new LabelEncoder<>(stringLabels);

/* 注意类别是按照在原始字符串数组中的顺序排列的 */
System.out.println(stringEncoder.getClasses()); //[Sunday, Monday, Tuesday]

for (Datum datum : data) {
    int classNumber = stringEncoder.encode(datum.getLabel);
    // 根据类别进行某项操作，即添加到List或Matrix中
}
```

注意，除了 `String` 类型，这种方法也适用于任何装箱类型，不过标签很可能采用适合于 `Short`、`Integer`、`Long`、`Boolean` 以及 `Character` 的值。例如，`Boolean` 标签可以是真 / 假的布尔值，`Character` 可以是 Y/N（是 / 否）或者 M/F（男 / 女），甚至是 T/F（真 / 假）。它完全取决于在所读取的数据文件中，原来的人是如何对标签进行编码的。标签不太可能采用浮点数形式，若是这样，大概是在处理回归问题，而不是分类问题（也就是说，将连续变量误认为是离散变量）。下一节将给出采用 `Integer` 类型标签的示例。

4.5.2　一位有效编码

在有些情况下，把多标签转换为多位二进制标签会更高效。这类似于把整数转换为二进制格式，只是要求一次只有一个位置是**有效的**（hot，等于 1）。例如，可以把 3 个字符串标签编码为整数，或者把每个字符串表示为二进制串中的一个位置。

```
Sunday  0  100
Monday  1  010
Tuesday 2  001
```

用 List 对标签编码时，采用下面的代码：

```
public class OneHotEncoder {

    private int numberOfClasses;

    public OneHotEncoder(int numberOfClasses) {
        this.numberOfClasses = numberOfClasses;
    }

    public int getNumberOfClasses() {
        return numberOfClasses;
    }

    public int[] encode(int label) {
        int[] oneHot = new int[numberOfClasses];
        oneHot[label] = 1;
        return oneHot;
    }

    public int decode(int[] oneHot) {
        return Arrays.binarySearch(oneHot, 1);
    }
}
```

若标签是字符串，则首先采用 LabelEncoder 实例把标签编码为整数，然后用 OneHotEncoder 实例把整数标签转换为一位有效编码。

```
String[] stringLabels = {"Sunday", "Monday", "Tuesday"};

LabelEncoder<String> stringEncoder = new LabelEncoder<>(stringLabels);

int numClasses = stringEncoder.getClasses.size();

OneHotEncoder oneHotEncoder = new oneHotEncoder(numClasses);

for (Datum datum : data) {
    int classNumber = stringEncoder.encode(datum.getLabel);
    int[] oneHot    = oneHotEncoder.encode(classNumber);
    // 根据类别进行某项操作，即添加到List或Matrix 中
}
```

反过来会怎么样呢？假设有个预测模型，它返回在学习过程中指定的类别。（通常，学习过程输出概率，但是此处可以假设已经把它们转换为类别。）首先，需要把一位有效输出转换为它的类别。接下来，需要把类别转换回它的原始标签，就像下面这样：

```
[1, 0, 0]
[0, 0, 1]
[1, 0, 0]
[0, 1, 0]
```

然后，需要把预测的输出从一位有效编码转换为原始标签。

```
for(Integer[] prediction: predictions) {
    int classLabel = oneHotEncoder.decode(prediction);
    String label = labelEncoder.decode(classLabel);
}

// 预测的标签是Sunday、Tuesday、Sunday、Monday
```

第5章

学习与预测

本章将研究数据的含义，以及数据如何驱动决策过程。从数据中学习可以获得知识，而运用知识可以对未来做出明智的预测。这就是数据科学存在的原因，即充分地学习数据，以便对新的数据做出预测。这可以像把数据分为多个组或者簇那么简单。经历一系列非常广泛的学习过程之后，机器学习最终将走向人工智能。学习分为监督型学习与无监督型学习两种。

一般来说，数据拥有变量 $\boldsymbol{X}$ 以及响应 $\boldsymbol{Y}$。我们的目标是采用 $\boldsymbol{X}$ 建立模型，以便预测若添上新的 $\boldsymbol{X}$ 时会发生什么。若有 $\boldsymbol{Y}$，则可以“监督”模型的建立。在许多场合，只有变量 $\boldsymbol{X}$，因此只能采用无监督的方式建立模型。典型的无监督型学习方法包括聚类，而监督型学习方法可以包括任何回归方法（例如线性回归）或分类方法，例如朴素贝叶斯、logistic 或者深度神经网络分类器。还有许多其他方法以及这些方法的交织，此处并不讨论所有方法，而只专注于少数几个最有用的方法。

5.1 学习算法

有几种学习算法广泛应用于大量技术中。具体而言，我们经常要用迭代学习过程来重复地对所寻找的模型参数进行优化或更新。有几种优化参数的方法，此处讨论梯度下降法。

5.1.1 迭代学习过程

学习模型的标准做法之一是遍历预测状态，并对其进行更新。回归、聚类以及期望最大化（EM）算法都获益于与迭代学习过程类似的形式。此处的策略是先创建一个包含所有迭代

学习过程样板的类。接下来，允许子类定义显式形式的预测以及参数更新方法：

```
public class IterativeLearningProcess {

    private boolean isConverged;
    private int numIterations;
    private int maxIterations;
    private double loss;
    private double tolerance;
    private int batchSize; // 若其值为0，则采用所有数据
    private LossFunction lossFunction;

    public IterativeLearningProcess(LossFunction lossFunction) {
        this.lossFunction = lossFunction;
        loss = 0;
        isConverged = false;
        numIterations = 0;
        maxIterations = 200;
        tolerance = 10E-6;
        batchSize = 100;
    }

    public void learn(RealMatrix input, RealMatrix target) {
        double priorLoss = tolerance;
        numIterations = 0;
        loss = 0;
        isConverged = false;
        Batch batch = new Batch(input, target);
        RealMatrix inputBatch;
        RealMatrix targetBatch;
        while(numIterations < maxIterations && !isConverged) {
            if(batchSize > 0 && batchSize < input.getRowDimension()) {
                batch.calcNextBatch(batchSize);
                inputBatch = batch.getInputBatch();
                targetBatch = batch.getTargetBatch();
            } else {
                inputBatch = input;
                targetBatch = target;
            }
            RealMatrix outputBatch = predict(inputBatch);
            loss = lossFunction.getMeanLoss(outputBatch, targetBatch);
            if(Math.abs(priorLoss - loss) < tolerance) {
                isConverged = true;
            } else {
                update(inputBatch, targetBatch, outputBatch);
                priorLoss = loss;
            }
            numIterations++;
        }
    }

    public RealMatrix predict(RealMatrix input) {
        throw new UnsupportedOperationException("Implement the predict method!");
    }
```

```
    public void update(RealMatrix input, RealMatrix target, RealMatrix output) {
        throw new UnsupportedOperationException("Implement the update method!");
    }

}
```

5.1.2 梯度下降优化方法

参数学习的一种方法是梯度下降法（一种一阶迭代优化算法）。该算法通过（利用误差）纠正学习过程，用递增地更新参数的方法来优化参数。术语**随机**（stochastic）意味着一次只添加单个点，而不是一次利用一整批数据。实际上，在迭代学习过程的每一步中，随机选择大约 100 个点的小批量数据是有益的。通常的想法是使损失函数最小化，从而通过下式对参数进行更新：

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

参数更新与目标函数 $f(\theta)$ 的梯度相关，如下式所示：

$$\Delta\theta \propto \nabla f(\theta)$$

对于深度网络，需要将误差通过网络进行反向传播，这一主题将在 5.4.3 节中详细讨论。

就本章而言，可以定义接口，在提供了特定梯度时返回参数更新。该接口包括矩阵形式与向量形式的方法签名。

```
public interface Optimizer {
    RealMatrix getWeightUpdate(RealMatrix weightGradient);
    RealVector getBiasUpdate(RealVector biasGradient);
}
```

梯度下降法中最常见的情况是从已有参数中减去缩放后的梯度，即下式：

$$\Delta\theta_t = -\eta\nabla f(\theta)_t$$

更新规则如下式所示：

$$\theta_{t+1} = \theta_t - \eta\nabla f(\theta)_t$$

随机梯度下降法（SGD，stochastic gradient descent）最常见的类型是采用学习速率对当前参数进行更新。

```
public class GradientDescent implements Optimizer {

    private double learningRate;

    public GradientDescent(double learningRate) {
```

```
        this.learningRate = learningRate;
    }

    @Override
    public RealMatrix getWeightUpdate(RealMatrix weightGradient) {
        return weightGradient.scalarMultiply(-1.0 * learningRate);
    }

    @Override
    public RealVector getBiasUpdate(RealVector biasGradient) {
        return biasGradient.mapMultiply(-1.0 * learningRate);
    }
}
```

这种优化算法的一种常见扩展是引入**动量**（momentum），当达到最优值时，它会放缓优化过程，以免超过正确参数。

$$\Delta\theta_t = \rho\Delta\theta_{t-1} - \eta\nabla f(\theta)_t$$

更新规则变为下式：

$$\theta_{t+1} = \theta_t + \rho\Delta\theta_{t-1} - \eta\nabla f(\theta)_t$$

可以看出，通过对 GradientDescent 类进行扩展，容易实现对动量的添加，从而为存储对权值与偏移量最近一次的更新做好准备，以用于计算下一次更新。注意在第一次迭代时，尚未存储之前的更新，因此要创建新集合，且初始化为零。

```
public class GradientDescentMomentum extends GradientDescent {

    private final double momentum;
    private RealMatrix priorWeightUpdate;
    private RealVector priorBiasUpdate;

    public GradientDescentMomentum(double learningRate, double momentum) {
        super(learningRate);
        this.momentum = momentum;
        priorWeightUpdate = null;
        priorBiasUpdate = null;
    }

    @Override
    public RealMatrix getWeightUpdate(RealMatrix weightGradient) {
        // 如果不存在，则创建与梯度大小相同的矩阵，其元素为0
        if(priorWeightUpdate == null) {
            priorWeightUpdate =
                new BlockRealMatrix(weightGradient.getRowDimension(),
                                    weightGradient.getColumnDimension());
        }
        RealMatrix update = priorWeightUpdate
                            .scalarMultiply(momentum)
                            .subtract(super.getWeightUpdate(weightGradient));
```

```
            priorWeightUpdate = update;
            return update;
        }

        @Override
        public RealVector getBiasUpdate(RealVector biasGradient) {
            if(priorBiasUpdate == null) {
                priorBiasUpdate = new ArrayRealVector(biasGradient.getDimension());
            }
            RealVector update = priorBiasUpdate
                                .mapMultiply(momentum)
                                .subtract(super.getBiasUpdate(biasGradient));
            priorBiasUpdate = update;
            return update;
        }
    }
```

梯度下降法是个持续发展且活跃的领域，采用引入动量的方法，容易扩展梯度下降法的能力，例如采用 ADAM 或 ADADELTA 算法。

5.2 评估学习过程

迭代过程可以无限期地运行下去。因此，必须设定允许迭代的最大次数，以阻止任何过程失去控制并永远计算下去。通常，迭代次数是 10^3~10^6 的量级，但没有一定的规则。如果已经满足特定标准，就可以早点结束迭代过程。这被称作**收敛**（convergence），也就是说迭代过程已经收敛到一个解，它看上去似乎是一个稳定的点（例如，自由参数不再以足够大的增量改变，使得迭代过程继续下去）。当然，有不止一种方式可以做到这些。尽管特定学习技术有其具体的收敛标准，但是没有通用方法。

5.2.1 损失函数最小化

损失函数（loss function）表示预测输出相对于目标输出的损失，也称作**成本函数**（cost function）或者**误差项**（error term）。给定单个输入向量 $\boldsymbol{x}$、输出向量 $\boldsymbol{y}$ 以及预测向量 $\hat{y}$，样本的损失记作$\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})$，损失函数的形式取决于输出数据所基于的统计分布。大多数情况下，p 维输出与预测之间的损失是每一维标量损失的累加和。

$$\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}}) = \sum_{p} \mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})$$

因为经常成批地处理数据，所以要计算整批数据的平均损失$\langle\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})\rangle$。当谈论损失函数的最小化时，是指把输入到学习算法中一批数据的平均损失最小化。在很多情况下，可以用损失函数的梯度$\nabla\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})$来对学习进行校正。通常，可以很容易地计算出损失相对于预测值的梯度$\frac{\partial\mathscr{L}}{\partial\hat{y}_i}$。于是，其指导思想是返回与输入形状相同的损失函数。



在一些教科书中，输出记作 t［**事实**（truth）或**目标**（target）］，预测记作 y。本书中输出记作 y，预测记作 $\hat{y}$。注意在这两种情况下，y 具有不同的含义。

损失函数的许多形式依赖于变量类型（连续、离散，或者两者都有）以及变量所基于的统计分布。不过，它们的共同点使得损失函数采用接口是理想的。把实现细节留给具体类的原因之一是，具体类要用到线性代数例程的优化算法。

```
public interface LossFunction {
    public double getSampleLoss(double predicted, double target);
    public double getSampleLoss(RealVector predicted, RealVector target);
    public double getMeanLoss(RealMatrix predicted, RealMatrix target);
    public double getSampleLossGradient(double predicted, double target);
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target);
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target);
}
```

1. 线性损失

也称作**绝对值损失**（absolute loss），**线性损失**（linear loss）是输出与预测之差的绝对值。

$$\mathscr{L}\left(y,\hat{y}\right)=\left|\hat{y}-y\right|$$

因为上式存在绝对值符号，所以计算梯度时容易出错，这一点不能忽略。

$$\frac{\partial\mathscr{L}(y,\hat{y})}{\partial\hat{y}}=\frac{\hat{y}-y}{\left|\hat{y}-y\right|}$$

当 $\hat{y}-y=0$ 时，由于 $\mathscr{L}$ 在该点不连续，所以梯度在该点没有定义。然而，在该点（$\hat{y}-y=0$）可以通过程序把梯度函数的值设定为 0，以避免发生 1/0 的异常。这样，梯度函数就只返回 –1、0 或 1。于是，理想的做法是使用数学函数 sign(x)，根据相应的输入值 $x<0$、$x=0$ 以及 $x>0$，它只返回 –1、0 或 1。

```
public class LinearLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return Math.abs(predicted - target);
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        return predicted.getL1Distance(target);
    }

    @Override
```

```
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            double dist = getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i));
            stats.addValue(dist);
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return Math.signum(predicted - target); // -1、0、1
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
        RealVector target) {
        return predicted.subtract(target).map(new Signum());
    }

    // 自己实现一个在稀疏矩阵上应用函数sign(x)的类SparseToSignum也不错
    // 只处理那些可迭代的元素
    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }

}
```

2. 二次损失

预测过程误差计算的一般做法是最小化整个数据集的距离指标，例如 L1 或 L2。对于特定的预测 – 目标对，二次误差如下式所示：

$$\mathscr{L}(y,\hat{y})=\frac{1}{2}(\hat{y}-y)^2$$

于是，样本损失梯度的元素如下式所示：

$$\frac{\partial\mathscr{L}}{\partial\hat{y}}=(\hat{y}-y)$$

二次损失函数的实现如下：

```
public class QuadraticLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        double diff = predicted - target;
        return 0.5 * diff * diff;
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double dist = predicted.getDistance(target);
        return 0.5 * dist * dist;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            double dist = getSampleLoss(predicted.getRowVector(i),
                                        target.getRowVector(i));
            stats.addValue(dist);
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return predicted - target;
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
        RealVector target) {
        return predicted.subtract(target);
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        return predicted.subtract(target);
    }
}
```

3. 交叉熵损失

交叉熵对于分类（例如 logistic 回归或者神经网络）是重要的。第 3 章讨论了交叉熵的起源。因为交叉熵表示两个样本之间的相似性，所以可以用它衡量已知值与预测值的一致性如何。对于学习算法，使 p 等同于已知值 y，q 等同于预测值 $\hat{y}$。设定损失等于交叉熵 $\mathscr{H}(p,q)$，从而 $\mathscr{L}(y,\hat{y})=\mathscr{H}(p,q)$，其中 $y_{ik}=p_{ik}$ 是目标（标签），$\hat{y}_{ik}=q_{ik}$ 是 K 个类别输出中的每个类别 k 的第 i 个预测值。于是，交叉熵（每个样本的损失）如下式所示：

$$\mathscr{L}(y,\hat{y})=-\sum_{k}^{K}y_k\log(\hat{y}_k)$$

交叉熵及其相关损失函数有下面几种常见形式。

4. 伯努利

对于伯努利型输出变量，已知的输出y是二元的，预测概率是$\hat{y}$，所产生的交叉熵损失如下式所示：

$$\mathscr{L}(y,\hat{y}) = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$

于是，样本损失梯度如下式所示：

$$\frac{\partial\mathscr{L}}{\partial\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$$

以下代码是伯努利交叉熵损失的实现：

```
public class CrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return -1.0 * (target * ((predicted>0)?FastMath.log(predicted):0)
            + (1.0 - target)*(predicted<1?FastMath.log(1.0-predicted):0));
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double loss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss += getSampleLoss(predicted.getEntry(i), target.getEntry(i));
        }
        return loss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        // 注意，若predicted = 0或1，将导致除法出错，因此应当保证永远不会发生这种情况
        return (predicted - target) / (predicted * (1 - predicted));
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
```

```
        RealVector loss = new ArrayRealVector(predicted.getDimension());
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss.setEntry(i, getSampleLossGradient(predicted.getEntry(i),
            target.getEntry(i)));
        }
        return loss;
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

上述表达式常用在 logistic 输出函数中。

5. 多项

若输出是多类别的（$k = 0, 1, 2, \cdots, K{-}1$），且通过一位有效编码转换为一系列的二元输出，则交叉熵损失是所有可能类别的总和。

$$\mathscr{L}(y,\hat{y}) = -\sum_{k} y_k \log(\hat{y}_k)$$

然而，在一位有效编码中，只有某个维度具有 $y = 1$，其余维度 $y = 0$（是稀疏矩阵）。因此，样本损失也是稀疏矩阵。考虑到这一点，理想情况下，可以简化计算。

样本损失梯度如下式所示：

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}}$$

因为损失矩阵的绝大多数元素是 0，所以只需要计算 $y = 1$ 位置上的梯度。这种形式主要用于归一化指数函数（softmax）输出函数中。

```
public class OneHotCrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        return predicted > 0 ? -1.0 * target * FastMath.log(predicted) : 0;
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double sampleLoss = 0.0;
```

```
        for (int i = 0; i < predicted.getDimension(); i++) {
            sampleLoss += getSampleLoss(predicted.getEntry(i),
                                        target.getEntry(i));
        }
        return sampleLoss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return -1.0 * target / predicted;
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        return target.ebeDivide(predicted).mapMultiplyToSelf(-1.0);
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return loss;
    }
}
```

6. 两点

当输出是二元的，但值为 –1 与 1，而不是 0 与 1 时，则可以在伯努利表达式中用 $y^* = (y+1)/2$ 与 $\hat{y}^* = (y+1)/2$ 替换，进行重新缩放。

$$\mathscr{L}\left(y^*, \hat{y}^*\right) = -\left(\left(\frac{y+1}{2}\right)\log\left(\frac{\hat{y}+1}{2}\right) + \left(\frac{1-y}{2}\right)\log\left(\frac{1-\hat{y}}{2}\right)\right)$$

样本损失梯度如下式所示：

$$\frac{\partial \mathscr{L}}{\partial \hat{y}} = \frac{\hat{y} - y}{1 - \hat{y}^2}$$

Java 代码如下所示：

```
public class TwoPointCrossEntropyLossFunction implements LossFunction {

    @Override
    public double getSampleLoss(double predicted, double target) {
        // 把-1~1的数转换为0~1的数
        double y = 0.5 * (predicted + 1);
        double t = 0.5 * (target + 1);
        return -1.0 * (t * ((y>0)?FastMath.log(y):0) +
                (1.0 - t)*(y<1?FastMath.log(1.0-y):0));
    }

    @Override
    public double getSampleLoss(RealVector predicted, RealVector target) {
        double loss = 0.0;
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss += getSampleLoss(predicted.getEntry(i), target.getEntry(i));
        }
        return loss;
    }

    @Override
    public double getMeanLoss(RealMatrix predicted, RealMatrix target) {
        SummaryStatistics stats = new SummaryStatistics();
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            stats.addValue(getSampleLoss(predicted.getRowVector(i),
            target.getRowVector(i)));
        }
        return stats.getMean();
    }

    @Override
    public double getSampleLossGradient(double predicted, double target) {
        return (predicted - target) / (1 - predicted * predicted);
    }

    @Override
    public RealVector getSampleLossGradient(RealVector predicted,
                                            RealVector target) {
        RealVector loss = new ArrayRealVector(predicted.getDimension());
        for (int i = 0; i < predicted.getDimension(); i++) {
            loss.setEntry(i, getSampleLossGradient(predicted.getEntry(i),
            target.getEntry(i)));
        }
        return loss;
    }

    @Override
    public RealMatrix getLossGradient(RealMatrix predicted, RealMatrix target) {
        RealMatrix loss = new Array2DRowRealMatrix(predicted.getRowDimension(),
        predicted.getColumnDimension());
        for (int i = 0; i < predicted.getRowDimension(); i++) {
            loss.setRowVector(i, getSampleLossGradient(predicted.getRowVector(i),
            target.getRowVector(i)));
```

```
        }
        return loss;
    }
}
```

这种形式的损失与双曲正切（tanh）激活函数兼容。

5.2.2　方差和的最小化

把数据分成多个组时，可以通过方差监测组内数据相对于其平均位置的分散程度。因为方差能够累加，所以可以定义 n 个组的指标 s，其中 σ_i^2 是每个组的方差。

$$s = \sum_{i=1}^{n} \sigma_i^2$$

当 s 减少时，表明总体误差也在减少。这对于聚类技术是有利的，例如 K 均值，它基于寻找每个簇的均值或形心点。

5.2.3　轮廓系数

无监督型学习方法，例如聚类，试图发现每个簇之内的点是如何紧密地聚在一起的。**轮廓系数**（silhouette coefficient）与任一给定簇之内的平均距离与该簇到与其最近簇之间的平均距离差值有关。下式中，轮廓分值 s 是每个样本的距离（定义见下式）s_i 的平均值。其中，a_i 为（某个类别的）样本与该类别中所有其他点的平均距离，b_i 为样本与下一个最近的簇中所有点的平均距离。

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

轮廓分值是所有样本轮廓系数的平均值：

$$s = \frac{1}{n} \sum_{i}^{n} s_i$$

轮廓分值在 −1~1 范围内，其中 −1 是不正确的聚类，1 是高度稠密的聚类，0 表示簇发生了重叠。随着簇变得稠密且有了很好的分离，s 将增大。监测过程的目标是寻找 s 的最大值。注意轮廓系数只在 $2 \leqslant n_{\text{labels}} \leqslant n_{\text{samples}} - 1$ 时有定义，下面是 Java 代码：

```
public class SilhouetteCoefficient {

    List<Cluster<DoublePoint>> clusters;
    double coefficient;
    int numClusters;
    int numSamples;
```

```
    public SilhouetteCoefficient(List<Cluster<DoublePoint>> clusters) {
        this.clusters = clusters;
        calculateMeanCoefficient();
    }

    private void calculateMeanCoefficient() {
        SummaryStatistics stats = new SummaryStatistics();
        int clusterNumber = 0;
        for (Cluster<DoublePoint> cluster : clusters) {
            for (DoublePoint point : cluster.getPoints()) {
                double s = calculateCoefficientForOnePoint(point, clusterNumber);
                stats.addValue(s);
            }
            clusterNumber++;
        }
        coefficient = stats.getMean();
    }

    private double calculateCoefficientForOnePoint(DoublePoint onePoint,
    int clusterLabel) {
        /* 所有其他点将与这个点进行比较 */
        RealVector vector = new ArrayRealVector(onePoint.getPoint());
        double a = 0;
        double b = Double.MAX_VALUE;
        int clusterNumber = 0;
        for (Cluster<DoublePoint> cluster : clusters) {
            SummaryStatistics clusterStats = new SummaryStatistics();
            for (DoublePoint otherPoint : cluster.getPoints()) {
                RealVector otherVector =
                    new ArrayRealVector(otherPoint.getPoint());
                double dist = vector.getDistance(otherVector);
                clusterStats.addValue(dist);
            }
            double avgDistance = clusterStats.getMean();
            if(clusterNumber==clusterLabel) {
                /* 已经引入了与自己距离为0的点 */
                /* 因此计算平均值时要减去这个点 */
                double n = new Long(clusterStats.getN()).doubleValue();
                double correction = n / (n - 1.0);
                a = correction * avgDistance;
            } else {
                b = Math.min(avgDistance, b);
            }
            clusterNumber++;
        }
        return (b-a) / Math.max(a, b);
    }
}
```

5.2.4　对数似然性

无监督型学习问题中，因为存在与每个输出预测相关的概率，所以可以利用对数似然性。

有个特例是本章的高斯聚类示例。对于这个期望最大化算法，对多维正态分布的混合做了优化以拟合数据。给定所采用的模型，每个数据点都有与之相关的概率密度 p_i，平均对数似然性可以用每个点概率对数的均值进行计算。

$$\mathscr{L}(\boldsymbol{p}) = \frac{1}{n}\sum_i \log(p_i)$$

于是，可以计算所有数据点的（平均）对数似然性$\langle\mathscr{L}(\boldsymbol{p})\rangle$。在高斯聚类示例中，通过 `MultivariateNormalMixtureExpectationMaximization.getLogLikelihood()` 方法可以直接得到这个参数。

5.2.5 分类器的准确率

如何知道分类器的实际准确率呢？二元分类方案有以下 4 种可能结果：

(1) 真阳性（TP），数据值与预测值均为 1；
(2) 真阴性（TN），数据值与预测值均为 0；
(3) 假阳性（FP），数据值是 0，但预测值为 1；
(4) 假阴性（FN），数据值是 1，但预测值为 0。

给定 4 种可能结果各自的计数，再结合其他参数，可以计算出分类器的准确率。

准确率（accuracy）可以按下式进行计算：

$$准确率 = \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$$

或者，鉴于分母是数据集总的行数 N，准确率表达式等价于下式：

$$准确率 = \frac{\mathrm{TP}+\mathrm{TN}}{N}$$

可以为每个维度计算准确率，准确率向量的平均值是分类器的平均准确率，也就是 Jaccard 分值。

在使用一位有效编码的特殊情况中，只需要真阳性，那么每一维的准确率表达式如下：

$$准确率 = \frac{\mathrm{TP}}{N_t}$$

N_t 是该维度总的分类计数（即该列中 1 的个数）。该分类器的准确率值计算如下：

$$准确率 = \frac{\Sigma_{\mathrm{TP}}}{N}$$

在实现中，有两种应用场景。第一种场景是一位有效编码。在另一种场景中，二元多标签输出是独立的。在后一种应用场景中，可以选择阈值（0~1 范围内），在这个点决定类别是 1 还是 0。在最一般的情况下，可以选择阈值为 0.5，将所有小于 0.5 的概率分类为 0，将大于等于 0.5 的概率分类为 1。这个类的应用示例将在 5.4 节中给出。

```
public class ClassifierAccuracy {

    RealMatrix predictions;
    RealMatrix targets;
    ProbabilityEncoder probabilityEncoder;
    RealVector classCount;

    public ClassifierAccuracy(RealMatrix predictions, RealMatrix targets) {
        this.predictions = predictions;
        this.targets = targets;
        probabilityEncoder = new ProbabilityEncoder();
        // 按每个维度针对二元类别进行计数
        classCount = new ArrayRealVector(targets.getColumnDimension());
        for (int i = 0; i < targets.getRowDimension(); i++) {
            classCount = classCount.add(targets.getRowVector(i));
        }
    }

    public RealVector getAccuracyPerDimension() {
        RealVector accuracy =
            new ArrayRealVector(predictions.getColumnDimension());
        for (int i = 0; i < predictions.getRowDimension(); i++) {
            RealVector binarized = probabilityEncoder.getOneHot(
                predictions.getRowVector(i));
            // 由于0*0 = 0、0*1 = 0、1*0 = 0 ，因此只有1*1 = 1是真阳性
            RealVector decision = binarized.ebeMultiply(targets.getRowVector(i));
            // 将TP计数添加到accuracy
            accuracy = accuracy.add(decision);
        }
        return accuracy.ebeDivide(classCount);
    }

    public double getAccuracy() {
        // 把每个维度的accuracy转换成计数
        // 然后求和，且除以总的行数
        return getAccuracyPerDimension().ebeMultiply(classCount).getL1Norm() /
        targets.getRowDimension();
    }

    // 实现Jaccard相似度值
    public RealVector getAccuracyPerDimension(double threshold) {
    // 假设多个输出不相关
        RealVector accuracy = new ArrayRealVector(targets.getColumnDimension());
        for (int i = 0; i < predictions.getRowDimension(); i++) {
            // 根据阈值把行向量二元化
            RealVector binarized = probabilityEncoder.getBinary(
            predictions.getRowVector(i), threshold);
            // 0-0 (TN)以及1-1 (TP) = 0，但是1-0 = 1以及0-1 = -1
```

```
            RealVector decision = binarized.subtract(
            targets.getRowVector(i)).map(new Abs()).mapMultiply(-1).mapAdd(1);
            // 将TP与TN计数添加到accuracy
            accuracy = accuracy.add(decision);
        }
        return accuracy.mapDivide((double) predictions.getRowDimension());
        // 对于给定的阈值，每个维度的准确率
    }

    public double getAccuracy(double threshold) {
        // accuracy向量的均值
        return getAccuracyPerDimension(threshold).getL1Norm() /
            targets.getColumnDimension();
    }
}
```

5.3 无监督型学习

若只有独立的变量，则必须在不借助于因变量（响应）或标签的情况下识别数据的模式。最常见的无监督型技术是聚类，所有聚类的目标是把数据点 $\boldsymbol{X}$ 划分为一系列的集合，可表示为 K 个集合，$S = S_1, \cdots, S_K$，其中集合个数小于数据点个数。通常，每个点 X_i 只属于其中的一个子集 S_k。然而，也可以指定每个点 X_i 分别属于各个集合的概率 $p(X_i) = p_1, p_2, \cdots, p_K$，其中各个概率的总和为 1。此处讨论**硬分配**（hard assignment）方法的两种变形，即 K 均值聚类与 DBSCAN 聚类，以及一种**软分配**（soft assignment）方法，即高斯混合模型。这些方法的假设、算法以及应用范围变化很大。然而，它们的结果通常是相同的，即把数据点 $\boldsymbol{X}$ 划分为一个或多个子集，或称作簇。

5.3.1 *K*均值聚类

K 均值聚类是聚类算法中最简单的形式，它采用直接分配方法寻找指定个数簇各自的形心。初始时，选定簇的个数 K（整数），用算法（或者随机）选定每个簇的形心位置 $\boldsymbol{\mu}_k$。对于点 $\boldsymbol{x}$，如果其与 $\boldsymbol{\mu}_k$ 的欧氏距离（也可以采用其他方式，但是通常是 L2）最小，则该点属于集合 S_k。于是，目标函数是将下式最小化：

$$\mathscr{L} = \sum_{k=1}^{K} \sum\nolimits_{\boldsymbol{x} \in S_k} \left\| \boldsymbol{x} - \boldsymbol{\mu}_k \right\|^2$$

然后，通过下式对形心位置（簇中所有点 $\boldsymbol{x}$ 的平均值）进行更新：

$$\boldsymbol{\mu}_k = \frac{1}{N} \sum_{\boldsymbol{x} \in S_k} \boldsymbol{x}$$

当目标函数不再改变，从而形心位置也不再改变时，就可以停止。如何知道怎样的簇个数是最优的？可以记录所有簇的方差总和，并调整簇的个数。当绘制方差之与簇个数的对比

图时，理想形状看起来像是曲棍球棒，图中弯曲明显处所对应的点就是簇的理想个数。

$$\sigma_K^2 = \sum_{k=1}^{K} \frac{1}{N_k - 1} \sum_{\boldsymbol{x} \in S_k} \left\| \boldsymbol{x} - \boldsymbol{\mu}_k \right\|^2$$

Apache Commons Math 所用算法是 K 均值聚类算法的改进版本，*k*-means++，它在选择随机初始点时做得更好。`KMeansPlusPlusClusterer<T>` 类的构造方法中有几个参数，但是只有其中之一是必需的，即所要寻找的簇个数。要进行聚类的数据必须是关于 `Clusterable` 点的 `List`。`DoublePoint` 类是对 `double` 型数组便利的封装，`DoublePoint` 类实现了 `Clusterable` 接口。在构造方法中，有个 `double` 型数组的参数：

```
double[][] rawData = ...

List<DoublePoint> data = new ArrayList<>();

for (double[] row : rawData) {
    data.add(new DoublePoint(row));
}

/* 所要寻找的簇个数 */
int numClusters = 1;

/* 基本的构造方法 */
KMeansPlusPlusClusterer<DoublePoint> kmpp =
    new KMeansPlusPlusClusterer<>(numClusters);

/* 进行聚类，且返回长度为numClusters的list */
List<CentroidCluster<DoublePoint>> results = kmpp.cluster(data);

/* 对关于Clusterable对象的列表进行迭代 */
for (CentroidCluster<DoublePoint> result : results) {

    DoublePoint centroid = (DoublePoint) result.getCenter();

    System.out.println(centroid); // DoublePoint具有toString()方法

    /* 也可以访问仅在这个簇中的所有点 */
    List<DoublePoint> clusterPoints = result.getPoints();

}
```

在 K 均值聚类方案中，需要对几种 `numClusters` 进行迭代，记录每个簇方差之和。因为方差能够累加，所以可以用其度量总体误差。理想情况下，需要使这个量最小化。此处，搜索不同簇个数进行迭代时，记录簇的方差。

```
/* 从1~5个簇中搜索 */
for (int i = 1; i < 5; i++) {

    KMeansPlusPlusClusterer<DoublePoint> kmpp = new KMeansPlusPlusClusterer<>(i);
    List<CentroidCluster<DoublePoint>> results = kmpp.cluster(data);
```

```
        /* 对于这种个数的簇，这是其方差之和 */
        SumOfClusterVariances<DoublePoint> clusterVar =
        new SumOfClusterVariances<>(new EuclideanDistance());

        for (CentroidCluster<DoublePoint> result : results) {
            DoublePoint centroid = (DoublePoint) result.getCenter());
        }
    }
```

对 K 均值聚类算法的一种改进方式是尝试几个起始点，再采用最佳结果——也就是说，使得误差最小。因为起始点是随机的，所以有时聚类算法会遇到错误，即使处理空簇的策略也无法处理。重复每个聚类尝试，并选择具有最佳结果的那个，是个不错的主意。`MultiKMeansPlusPlusClusterer<T>` 类进行 `numTrials` 次相同的聚类操作，只采用最佳结果。可以把这些操作与前述代码进行结合。

```
    /* 对于每种聚类尝试，重复10次，选择最佳结果 */
    int numTrials = 10;

    /* 从1~5个簇中搜索 */
    for (int i = 1; i < 5; i++) {

        /* 仍然需要创建KMeansPlusPlusClusterer的实例…… */
        KMeansPlusPlusClusterer<DoublePoint> kmpp = new KMeansPlusPlusClusterer<>(i);

        /* ……并把它传递给MultiKMeansPlusPlusClusterer的构造方法 */
        MultiKMeansPlusPlusClusterer<DoublePoint> multiKMPP =
            new MultiKMeansPlusPlusClusterer<>(kmpp, numTrials);

        /* 注意此处是对multiKMPP进行聚类，而不是对kmpp */
        List<CentroidCluster<DoublePoint>> results = multikKMPP.cluster(data);

        /* 对于这种个数的簇，这是其方差之和 */
        SumOfClusterVariances<DoublePoint> clusterVar =
            new SumOfClusterVariances<>(new EuclideanDistance());

        /* i个簇的方差数值之和 */
        double score = clusterVar.score(results)

        /* 最佳形心位置 */
        for (CentroidCluster<DoublePoint> result : results) {
            DoublePoint centroid = (DoublePoint) result.getCenter());
        }
    }
```

5.3.2 DBSCAN

当簇的形状不规则时，会如何呢？当簇互相重叠时，又会如何呢？ DBSCAN（density-based spatial clustering of applications with noise，有噪声应用的基于密度的空间聚类）算法可以很好地发现难以分类的簇。它不预设簇的个数，但是它本身会对现有簇的个数进行优化。所

需的输入参数是最大采集半径以及每个簇中点的最小个数。实现代码如下：

```
/* 构造方法的参数有eps以及minPts */
double eps = 2.0;
int minPts = 3;
DBSCANClusterer clusterer = new DBSCANClusterer(eps, minPts);
List<Cluster<DoublePoint>> results = clusterer.cluster(data);
```

注意，因为不规则形状簇的形心位置可能没有意义，所以与前述 *k*-means++ 算法不同，DBSCAN 并不返回 `CentroidCluster` 类型。不过，可以直接访问已经聚类的点，并对其做进一步处理。但是要注意，如果算法不能发现任何簇，那么 `List<Cluster<T>>` 实例将包含大小为 0 的空 `List`。

```
if(results.isEmpty()) {
    System.out.println("No clusters were found");
} else {

    for (Cluster<DoublePoint> result : results) {
        /* 每个簇的点在此处 */
        List<DoublePoint> points = result.getPoints();
        System.out.println(points.size());
        // 根据每个簇中的点进行相应处理
    }
}
```

在这个例子中，创建了 4 个随机的多元（二维）一般簇。需要注意的是，其中有两个簇靠得非常近，几乎要互相接触，甚至可以把它们看作带尖角的簇。这表明在 DBSCAN 算法中需要进行折中。

在这种情况下，需要设置足够小的采集半径（$\epsilon = 0.225$），以便于监测到分离的簇，但是会有异常值。此处，较大的半径（$\epsilon = 0.8$）会把最左边两个簇合二为一，几乎没有异常值。随着ϵ的减小，检测簇的分辨率会提高，但是也增加了异常值的似然性。在更高维度的空间中，簇之间彼此接近的可能性不大，因此这不是什么问题。图 5-1 中给出了适合采用 DBSCAN 算法的 4 个高斯簇示例。[1]

注 1：读者可访问本书图灵社区页面（https://www.ituring.com.cn/book/2082），在右侧的“随书下载”中查看图 5-1 的彩色图片。——编者注

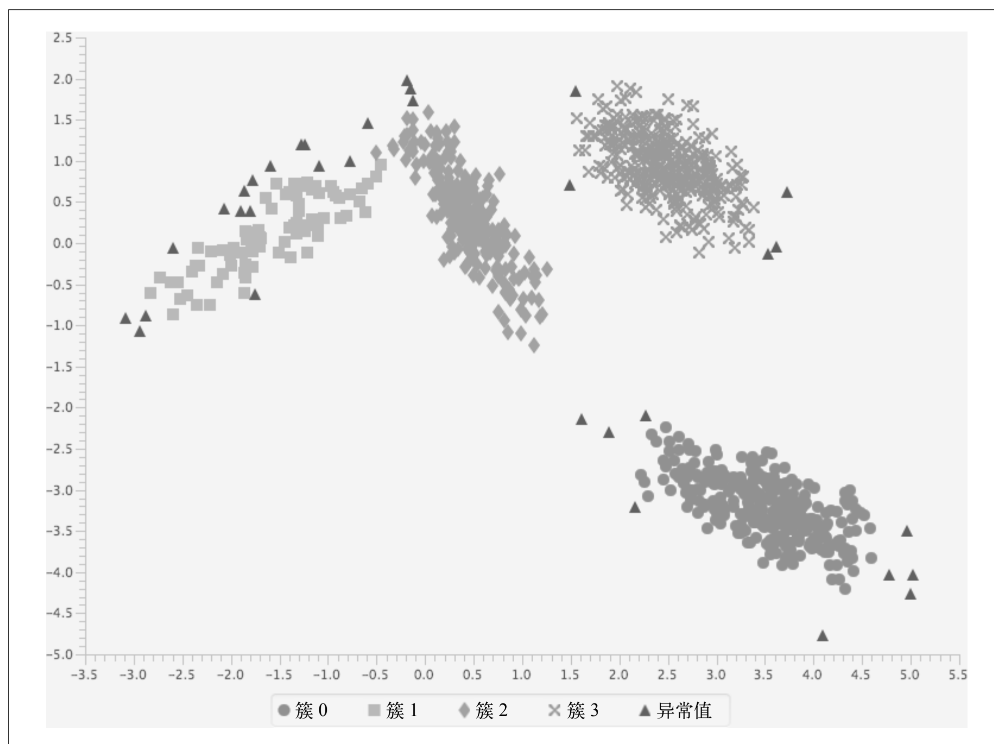


图 5-1：采用 DBSCAN 对 4 个高斯簇的模拟

1. 处理异常值

DBSCAN 算法非常适合处理异常值。但是，如何访问这些异常值呢？遗憾的是，目前的 Math 实现中不允许访问 DBSCAN 算法中标记为噪声的点。不过，可以像下面这样对其进行记录：

```
/* 要记录异常值 */
// 注意需要全新列表，但是不要引用相同的对象
// 例如，outliers = data不是一种好的做法

// List<DoublePoint> outliers = data; // 也会从data中移除点

List<DoublePoint> outliers = new ArrayList<>();

for (DoublePoint dp : data) {
    outliers.add(new DoublePoint(dp.getPoint()));
}
```

于是，对结果簇进行迭代时，可以从完整的数据集中移除每个簇，当移除这些簇之后，剩下的数据将成为异常值。

```
for (Cluster<DoublePoint> result : results) {

    /* 每个簇的点在此处 */
    List<DoublePoint> points = result.getPoints();


    /* 从data的副本outliers中移除这些簇的点,
       当所有簇的点都被移除之后，outliers将只包含异常值
    */

    outliers.removeAll(points);
}

// 现在，列表outliers中只包含那些不在任何簇中出现的点
```

2. 对采集半径以及最少点数进行优化

在二维问题中，容易观测采集半径，但是如何知道它是最优的？显然，这完全取决并依赖于应用场景。一般来说，最少点数应该满足下式[2]：

$$n_{\min} \geqslant p + 1$$

因此，在二维问题中，每个簇中至少需要三个点。采集半径 ϵ 可以根据 k 距离图（k-distance graph）曲棍球棒的明显弯曲处进行估计。采用轮廓分值作为指标，可以进行网格搜索，从而得到最少点数以及采集半径。首先，计算每个样本的轮廓系数 s。其中，a 表示样本与类别中所有其他点之间的平均距离，b 表示样本与下一个最近的簇中所有点的平均距离。

$$s = \frac{b - a}{\max(a, b)}$$

于是，轮廓分值就是所有样本的轮廓系数的平均值。轮廓分值在 −1~1 范围内，其中 −1 是不正确的聚类，1 是高度稠密的聚类，0 表示簇发生了重叠。随着簇变得稠密以及有了很好的分离，s 将增大。就像前述的 K 均值聚类那样，可以调整ϵ的值，并输出轮廓分值。

```
double[] epsVals = {0.15, 0.16, 0.17, 0.18, 0.19, 0.20,
                    0.21, 0.22, 0.23, 0.24, 0.25};

for (double epsVal : epsVals) {

    DBSCANClusterer clusterer = new DBSCANClusterer(epsVal, minPts);
    List<Cluster<DoublePoint>> results = clusterer.cluster(dbExam.clusterPoints);

    if(results.isEmpty()) {

        System.out.println("No clusters where found");

    } else {
```

注 2：此处 p 是数据的维度。——译者注

```
        SilhouetteCoefficient s = new SilhouetteCoefficient(results);
        System.out.println("eps = " + epsVal +
                          " numClusters = " + results.size() +
                          " s = " + s.getCoefficient());
    }
}
```

这将产生下列输出：

```
eps = 0.15 numClusters = 7 s = 0.54765
eps = 0.16 numClusters = 7 s = 0.53424
eps = 0.17 numClusters = 7 s = 0.53311
eps = 0.18 numClusters = 6 s = 0.68734
eps = 0.19 numClusters = 6 s = 0.68342
eps = 0.20 numClusters = 6 s = 0.67743
eps = 0.21 numClusters = 5 s = 0.68348
eps = 0.22 numClusters = 4 s = 0.70073 // 最好的
eps = 0.23 numClusters = 3 s = 0.68861
eps = 0.24 numClusters = 3 s = 0.68766
eps = 0.25 numClusters = 3 s = 0.68571
```

当$\epsilon = 0.22$时，轮廓分值出现了凸出点，$s = 0.7$，表明理想的ϵ接近于 0.22。对于这个特定的ϵ，DBSCAN 例程也收敛到四个簇，这是所模拟的簇个数。当然，在实际情况中，并不能预先知道簇的个数。但是这个示例确实表明，若簇的个数正确，从而得到正确的ϵ值，则 s 应当接近于最大值 1。

3. DBSCAN 的推论

与 K 均值算法不同，DBSCAN 算法不是用来预测新数据点的归属，而是用于对数据进行划分，以供进一步使用。如果需要基于 DBSCAN 的预测模型，那么可以分配类别值给已经聚类的数据点，然后尝试分类方案，例如高斯混合、朴素贝叶斯以及其他方案。

5.3.3　高斯混合

与 DBSCAN 的概念类似，高斯混合模型是基于点的密度进行聚类，但采用多维正态分布 $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$，因为它由均值与协方差构成。接近于平均位置的数据点具有属于该簇的最高概率，随着数据点远离平均位置，属于该簇的概率下降直至 0。

1. 高斯混合模型

高斯混合模型在数学上表示为 k 维高斯分布的加权混合（见第 3 章的讨论）。

$$f(\boldsymbol{x}) = \sum_{i=1}^{k} \alpha_i \mathcal{N}(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

此处的权值满足等式 $\sum_i^k \alpha_i = 1$。必须创建关于 Pair 对象的 List，Pair 的第一个成员是权值，第二个成员是分布自身。

```
List<Pair<Double, MultivariateNormalDistribution>> mixture = new ArrayList<>();

/* 第一个混合组件 */
double alphaOne = 0.70;
double[] meansOne = {0.0, 0.0};
double[][] covOne = {{1.0, 0.0},{0.0, 1.0}};
MultivariateNormalDistribution distOne =
    new MultivariateNormalDistribution(meansOne, covOne);
Pair pairOne = new Pair(alphaOne, distOne);
mixture.add(pairOne);

/* 第二个混合组件 */
double alphaTwo = 0.30;
double[] meansTwo = {5.0, 5.0};
double[][] covTwo = {{1.0, 0.0},{0.0, 1.0}};
MultivariateNormalDistribution distTwo =
    new MultivariateNormalDistribution(meansTwo, covTwo);
Pair pairTwo = new Pair(alphaTwo, distTwo);
mixture.add(pairTwo);

/* 把Pair对象的列表添加到混合模型中，并对点进行采样 */
MixtureMultivariateNormalDistribution dist =
    new MixtureMultivariateNormalDistribution(mixture);

/* 不需要种子，但是如果想重用相同的数据，那么这样做是有帮助的 */
dist.reseedRandomGenerator(0L);

/* 从mixture中生成1000个随机数据点 */
double[][] data = dist.sample(1000);
```

注意，混合分布模型中的样本数据并不记录所采样的数据点来自哪个组件。换句话说，不能够辨别每个样本的数据点属于哪个 `MultivariateNormal`。如果需要这个功能，那么可以总是从独立的分布中采样，然后再把它们加在一起。

为了测试而创建混合模型会是单调乏味的，且充满了问题。如果不是从已有实际数据创建数据集，那么最好是尝试那些已经避开一些已知问题的模拟数据。附录 A 中给出了产生随机混合模型的一种方法。图 5-2 中绘制了多维高斯混合模型，其中有关于二维数据的两个簇。

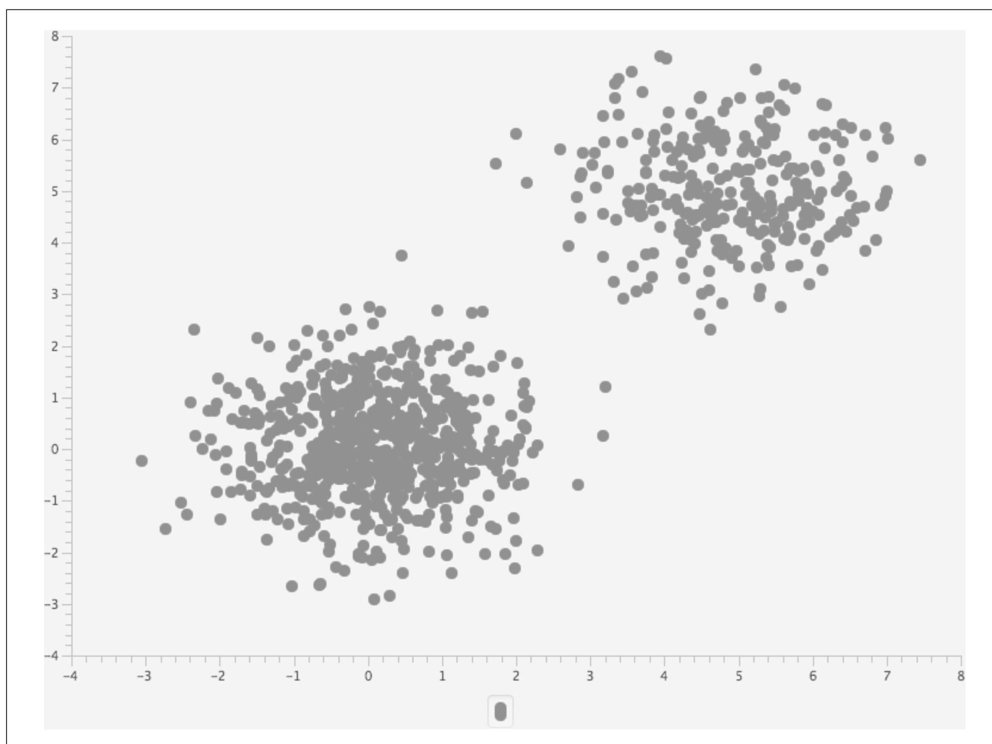


图 5-2：二维数据的高斯簇

数据可以用下面的示例代码生成：

```
int dimension = 5;
int numClusters = 7;
double boxSize = 10;
long seed = 0L;
int numPoints = 10000;

/* 这个数据集见附录A */
MultiNormalMixtureDataset mnd = new MultiNormalMixtureDataset(dimension);
mnd.createRandomMixtureModel(numClusters, boxSize, 0L);
double[][] data = mnd.getSimulatedData(numPoints);
```

2. 用 EM 算法拟合

期望最大化算法（EM）应用在许多其他场合。最本质的问题是，所选参数正确的最大似然性是多少？给定特定容限，进行迭代，直到这些参数不再改变为止。此处需要提供初始猜测，即混合模型是什么。采用前一节的方法，用已知组件，可以创建混合模型。不过，给定数据集以及簇个数作为输入，可用静态方法 `MultivariateNormalMixtureExpectationMaximization.estimate(data, numClusters)` 来估计起始点。

```
MultivariateNormalMixtureExpectationMaximization mixEM =
    new MultivariateNormalMixtureExpectationMaximization(data);

/* 需要猜测从哪里开始 */
MixtureMultivariateNormalDistribution initialMixture =
    MultivariateNormalMixtureExpectationMaximization.estimate(data, numClusters);

/* 进行拟合 */
mixEM.fit(initialMixture);

/* 这是拟合模型 */
MixtureMultivariateNormalDistribution fittedModel = mixEM.getFittedModel();

for (Pair<Double, MultivariateNormalDistribution> pair :
    fittedModel.getComponents()) {
    System.out.println("************ cluster *****************");
    System.out.println("alpha: " + pair.getFirst());
    System.out.println("means: " + new ArrayRealVector(
        pair.getSecond().getMeans()));
    System.out.println("covar: " + pair.getSecond().getCovariances());
}
```

3. 优化簇的个数

正如在 K 均值聚类中，我们想要知道描述数据所需的最优簇个数。不过在这种情况下，每个数据点按照限定的概率属于所有簇（即软分配）。如何知道簇的个数满足要求？以小的数值开始（例如 2），然后逐步前进，计算每次尝试的对数似然性。为了操作更容易，可以绘制损失（对数似然性的负值）的图，当它有希望降至零时要留意。实际上，它从来不会降至零，不过当损失在某种程度上不变时就停止这个过程。通常，簇的最佳个数是曲棍球棒形状图像的明显弯曲处。代码如下：

```
MultivariateNormalMixtureExpectationMaximization mixEM =
    new MultivariateNormalMixtureExpectationMaximization(data);

int minNumClusters = 2;
int maxNumClusters = 10;

for(int i = minNumCluster; i <= maxNumClusters; i++) {

    /* 需要猜测从哪里开始 */
    MixtureMultivariateNormalDistribution initialMixture =
        MultivariateNormalMixtureExpectationMaximization.estimate(data, i);

    /* 进行拟合 */
    mixEM.fit(initialMixture);

    /* 这是拟合模型 */
    MixtureMultivariateNormalDistribution fittedModel = mixEM.getFittedModel();

    /* 打印输出对数似然性 */
    System.out.println(i + " ll: " + mixEM.getLogLikelihood());
}
```

输出如下：

```
2 ll: -6.370643787350135
3 ll: -5.907864928786343
4 ll: -5.5789246749261014
5 ll: -5.366040927493185
6 ll: -5.093391683217386
7 ll: -5.1934910558216165
8 ll: -4.984837507547836
9 ll: -4.9817765145490664
10 ll: -4.981307556011888
```

对以上数据进行绘制，将显示出典型的曲棍球棒形状，拐点位于 `numClusters = 7`，即所模拟的簇个数。注意，我们也可以把对数似然性存储在数组中，并且使结果适合放在 `List` 中，以供随后的程序使用。图 5-3 中给出了对数似然性损失相对于簇个数的绘图。注意损失急剧下降，在 7 个簇时发生了弯曲，这正是模拟数据集中簇最初的个数。

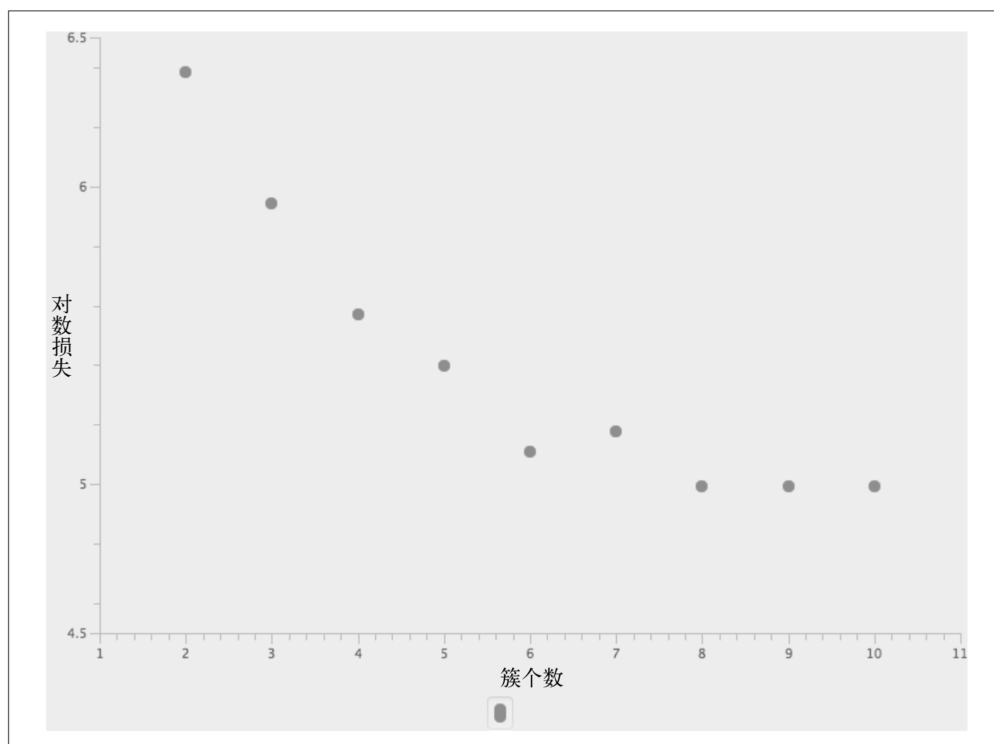


图 5-3：7 个五维数据簇的对数损失

5.4 监督型学习

给定数值变量 X 及潜在的非数值响应 Y，如何表示学习与预测之间的数学模型？回想一

下，线性回归模型要求 $\boldsymbol{X}$ 及 $\boldsymbol{Y}$ 都是连续变量（例如实数）。即使 $\boldsymbol{Y}$ 包含多个 0 或多个 1（以及任何其他整数），线性回归也很可能会失败。

这里讨论为常见应用场景专门设计的方法，这些应用场景搜集数值数据作为变量及其相应的标签。多数分类方案适合于多维变量 $\boldsymbol{X}$ 以及单维类别 $\boldsymbol{Y}$。然而，采用类似于线性回归中多响应模型的方式，包括神经网络在内的几种技术可以处理多输出类别 $\boldsymbol{Y}$。

5.4.1 朴素贝叶斯

朴素贝叶斯可能是最基本的分类方案，从逻辑上来说，它是聚类之后的下一步。记得在聚类中，目标是把数据分离或分类成为不同的组。然后可以独立地考察每个组，试图获得关于该组的某些信息，例如其形心位置、方差，或者任何其他统计指标。

在朴素贝叶斯分类方案中，为每个标签类型把数据分为若干组（类别）。然后从每个组的变量中获得某些信息，这取决于变量类型。例如，若变量是实数，则可以假设数据的每个维度（变量）是服从正态分布的样本。

对于整数数据（计数），可以假设它服从多项式分布。若数据是二元的（0 或 1），则可以假设它是服从伯努利分布的数据集。这样就可以估计统计量，例如只属于所标记类别的每个数据集的均值以及方差。注意，不同于更复杂的分类方案，在任何计算或者误差传播中，从不使用标签本身。标签仅用于把数据分为不同的组。

根据贝叶斯定理（后验概率 = 先验概率 × 似然性 / 证据），联合概率等于先验概率 × 似然性。在这里，证据（evidence）是所有类别的联合概率之和。对于 K 个类别的集合，其中，$k = \{1, 2, \cdots, K\}$，给定输入向量 $\boldsymbol{x}$，特定类别 k 的概率计算公式如下式：

$$p(k \mid \boldsymbol{x}) = \frac{p(k)p(\boldsymbol{x} \mid k)}{p(\boldsymbol{x})}$$

此处，朴素的独立性假设允许把似然性表示为 n 维变量 $\boldsymbol{x}$ 每个维度的概率之积：

$$p(\boldsymbol{x} \mid k) = p(x_1 \mid k)p(x_2 \mid k)\cdots p(x_n \mid k)$$

用下式可以更加紧凑地表示：

$$p(\boldsymbol{x} \mid k) = \prod_{i=1}^{n} p(x_i \mid k)$$

归一化是把分子的所有项进行求和：

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} p(k)p(\boldsymbol{x} \mid k)$$

每个类别的概率是它发生的次数除以总数，即 $p(k) = n_k/N$。此处采用每个类别 k 在每个特征 x_i 上的乘积。$p(x_i | C_K)$ 的形式，是根据关于数据的假设所选取的概率密度函数。接下来的章节会讨论正态分布、多项式分布以及伯努利分布。



注意，若 $p(x_i | k) = 0$，则整个表达式将成为 $p(k | \boldsymbol{x}) = 0$。对于一些条件概率模型，例如高斯分布或伯努利分布，这种情况永远不会发生。但是对于多项式分布，这可能会发生，因此引入小的因子 α 来避免这种情况。

为每个类别计算后验概率之后，则贝叶斯分类器就是关于后验概率的决策规则，此处采用最大的位置作为最可能的类别。

$$\hat{k} = \underset{k \in \{1,2,\cdots,K\}}{\arg\max}\; p(k | \boldsymbol{x})$$

可以为所有类型使用相同的类别，因为模型训练依赖于量的类型，这些量可以很容易地按照每个类别用 MultivariateSummaryStatistics 进行累计。采用一种策略模式来实现所需要的任意一种条件概率类型，并且把它直接传递给构造方法。

```
public class NaiveBayes {

    Map <Integer, MultivariateSummaryStatistics> statistics;
    ConditionalProbabilityEstimator conditionalProbabilityEstimator;
    int numberOfPoints; // 训练模型所用的数据点总数

    public NaiveBayes(
        ConditionalProbabilityEstimator conditionalProbabilityEstimator) {
        statistics = new HashMap<>();
        this.conditionalProbabilityEstimator = conditionalProbabilityEstimator;
        numberOfPoints = 0;
    }

    public void learn(RealMatrix input, RealMatrix target) {
        // 若numTargetCols == 1，则采用多类别，例如0、1、2、3
        // 否则采用一位有效编码，例如1000、0100、0010、0001
        numberOfPoints += input.getRowDimension();
        for (int i = 0; i < input.getRowDimension(); i++) {
            double[] rowData = input.getRow(i);
            int label;
            if (target.getColumnDimension()==1) {
                label = new Double(target.getEntry(i, 0)).intValue();
            } else {
                label = target.getRowVector(i).getMaxIndex();
            }

            if(!statistics.containsKey(label)) {
                statistics.put(label, new MultivariateSummaryStatistics(
                rowData.length, true));
            }
            statistics.get(label).addValue(rowData);
```

```
        }
    }

    public RealMatrix predict(RealMatrix input) {

        int numRows = input.getRowDimension();
        int numCols = statistics.size();
        RealMatrix predictions = new Array2DRowRealMatrix(numRows, numCols);

        for (int i = 0; i < numRows; i++) {
            double[] rowData = input.getRow(i);
            double[] probs = new double[numCols];
            double sumProbs = 0;
            for (Map.Entry<Integer, MultivariateSummaryStatistics> entrySet :
            statistics.entrySet()) {

                Integer classNumber = entrySet.getKey();
                MultivariateSummaryStatistics mss = entrySet.getValue();

                /* 先验概率(n_k/N)，即(该类别中的点数 / 总点数) */
                double prob = new Long(mss.getN()).doubleValue()/numberOfPoints;

                /* 取决于类型……例如高斯分布、多项式分布，或伯努利分布 */
                prob *= conditionalProbabilityEstimator.getProbability(mss,
                                                                       rowData);

                probs[classNumber] = prob;
                sumProbs += prob;
            }

            /* 概率的L1范数归一化 */
            for (int j = 0; j < numCols; j++) {
                probs[j] /= sumProbs;
            }
            predictions.setRow(i, probs);
        }
        return predictions;
    }
}
```

接下来只需要一个接口来指出条件概率的形式。

```
public interface ConditionalProbabilityEstimator {
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features);
}
```

接下来将讨论 3 种朴素贝叶斯分类器，其中每个分类器都实现了 `NaiveBayes` 类中所使用的 `ConditionalProbabilityEstimator` 接口。

1. 高斯朴素贝叶斯分类器

若特征是连续变量，则可以使用高斯朴素贝叶斯分类器。

$$p(\boldsymbol{x} \mid k) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_{ki}} \exp\left(-\frac{(x_i - \mu_{ki})^2}{2\sigma_{ki}^2}\right)$$

然后，可以像下面这样实现类：

```
import org.apache.commons.math3.distribution.NormalDistribution;
import org.apache.commons.math3.stat.descriptive.MultivariateSummaryStatistics;

public class GaussianConditionalProbabilityEstimator
implements ConditionalProbabilityEstimator{

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        double[] means = mss.getMean();
        double[] stds = mss.getStandardDeviation();
        double prob = 1.0;
        for (int i = 0; i < features.length; i++) {
            prob *= new NormalDistribution(means[i], stds[i])
                .density(features[i]);
        }
        return prob;
    }

}
```

可以像下面这样对其进行测试：

```
double[][] features = {{6, 180, 12},{5.92, 190, 11}, {5.58, 170, 12},
                       {5.92, 165, 10}, {5, 100, 6}, {5.5, 150, 8},
                       {5.42, 130, 7}, {5.75, 150, 9}};
String[] labels = {"male", "male", "male", "male",
                   "female", "female", "female", "female"};
NaiveBayes nb = new NaiveBayes(new GaussianConditionalProbabilityEstimator());
nb.train(features, labels);

double[] test = {6, 130, 8};
String inference = nb.inference(test); // female
```

这将产生正确的结果，即 `female`。

2. 多项朴素贝叶斯分类器

特征是整型值，例如计数值。然而，连续的特征，例如 TFIDF 也同样适用。对于类别 k，观测到任何特征向量 $\boldsymbol{x}$ 的似然性，如下式：

$$p(\boldsymbol{x} \mid k) = \frac{(\Sigma_{i=1}^{n} x_i)!}{\Pi_{i=1}^{n} x_i!} \prod_{i=1}^{n} p_{ki}^{x_i}$$

注意，由于等号右边前面的项仅依赖于输入向量 $\boldsymbol{x}$，因此对于每个计算，$p(\boldsymbol{x} \mid k)$ 都是相同

的。幸运的是，在最终的归一化表达式 $p(\boldsymbol{x} \mid k)$ 中会消去这个计算密集的项，从而能够使用简洁得多的表达式。

$$p(\boldsymbol{x} \mid k) = \prod_{i=1}^{n} p_{ki}^{x_i}$$

可以很容易地计算所需要的概率 $p_{ki} = N_{ik}/N_k$，其中 N_{ik} 是类别 k 中每个特征值的总和，N_k 是类别 k 中所有特征值的总和。在估计条件概率时，任何零都会抵消整个计算。因此，在估计概率时，引入小的附加因子 α 是有用的：当 $0 < \alpha < 1$ 时，称作 **Lidstone 平滑**；当 $\alpha = 1$ 时，称作**拉普拉斯**平滑。由于用 L1 范数对分子进行归一化，因此因子 n 正是特征向量的维数。

$$p_{ki} = \frac{N_{ik} + \alpha}{N_k + \alpha n}$$

最后的表达式如下式：

$$p(\boldsymbol{x} \mid k) = \prod_{i=1}^{n} \left(\frac{N_{ik} + \alpha}{N_k + \alpha n} \right)^{x_i}$$

对于大的 x_i（例如大量的单词），在数值上将很难处理。可以用求对数的方法解决这个问题，并采用等式 $z = \exp(\ln(z))$ 还原回来。前面的表达式可以写作下式：

$$p(\boldsymbol{x} \mid k) = \exp\left(\sum_{i=1}^{n} x_i \ln\left(\frac{N_{ik} + \alpha}{N_k + \alpha n} \right) \right)$$

在采用这种策略进行实现时，在构造方法中指定平滑系数。注意采用对数实现是为了避免数值不稳定。在构造方法中添加声明，要求平滑系数常量 α 满足关系式 $0 < \alpha \leqslant 1$ 是明智的。

```
public class MultinomialConditionalProbabilityEstimator
        implements ConditionalProbabilityEstimator {

    private double alpha;

    public MultinomialConditionalProbabilityEstimator(double alpha) {
        this.alpha = alpha; // Lidstone平滑：0 < α ≤ 1
    }

    public MultinomialConditionalProbabilityEstimator() {
        this(1); // 拉普拉斯平滑
    }

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                double[] features) {
```

```
        int n = features.length;
        double prob = 0;
        double[] sum = mss.getSum(); // 该类别xi总和的数组
        double total = 0.0; // 所有特征的总数
        for (int i = 0; i < n; i++) {
            total += sum[i];
        }
        for (int i = 0; i < n; i++) {
            prob += features[i] * Math.log((sum[i] + alpha) /(total+alpha*n));
        }
        return Math.exp(prob);
    }

}
```

3. 伯努利朴素贝叶斯分类器

特征是二元值，例如占用状态。每个特征的概率是该列的平均值。对于输入特征，则可以计算下面的概率：

$$p(\boldsymbol{x} \mid k) = \prod_{i=1}^{n} (p_{ki} x_i + (1 - p_{ki})(1 - x_i))$$

换句话说，若输入特征是 1，则该特征的概率是该列的平均值。如果输入特征是 0，则该特征的概率是 1，即该列的平均值。像下面这样实现伯努利条件概率：

```
public class BernoulliConditionalProbabilityEstimator
    implements ConditionalProbabilityEstimator {

    @Override
    public double getProbability(MultivariateSummaryStatistics mss,
                                 double[] features) {
        int n = features.length;
        double[] means = mss.getMean();
        // 这实际上是每个特征的概率，例如(计数/总数)
        double prob = 1.0;
        for (int i = 0; i < n; i++) {
            // 若xi = 1，则采用p；若xi = 0，则采用1-p，但此处xi是double型数据
            prob *= (features[i] > 0.0) ? means[i] : 1-means[i];
        }
        return prob;
    }
}
```

4. Iris 示例

采用高斯条件概率估计器来测验 Iris 数据集。

```
Iris iris = new Iris();
MatrixResampler mr = new MatrixResampler(iris.getFeatures(), iris.getLabels());
mr.calculateTestTrainSplit(0.4, 0L);

NaiveBayes nb = new NaiveBayes(new GaussianConditionalProbabilityEstimator());
```

```
nb.learn(mr.getTrainingFeatures(), mr.getTrainingLabels());

RealMatrix predictions = nb.predict(mr.getTestingFeatures());

ClassifierAccuracy acc = new ClassifierAccuracy(predictions,
                                                mr.getTestingLabels());
System.out.println(acc.getAccuracyPerDimension()); // {1; 1; 0.9642857143}
System.out.println(acc.getAccuracy()); // 0.9833333333333333
```

5.4.2 线性模型

如果对数据集 $\boldsymbol{X}$ 进行旋转、平移、缩放，那么能否通过函数映射将其与输出 $\boldsymbol{Y}$ 联系起来？一般来说，在这些操作试图解决的问题中，输入矩阵 $\boldsymbol{X}$ 是数据，$\boldsymbol{W}$ 与 $\boldsymbol{b}$ 是要进行优化的自由参数。采用第 2 章的表示法，对于加权的输入矩阵以及截距项 $\boldsymbol{Z} = \boldsymbol{XW} + \boldsymbol{hb}^{\mathrm{T}}$，对 $\boldsymbol{Z}$ 的每个元素应用函数 $\varphi(\boldsymbol{Z})$，按下式计算预测矩阵 $\hat{\boldsymbol{Y}}$：

$$\hat{\boldsymbol{Y}} = \varphi(\boldsymbol{XW} + \boldsymbol{hb}^{\mathrm{T}})$$

可以把线性模型看作是关于输入 $\boldsymbol{X}$ 以及预测输出 $\hat{\boldsymbol{Y}}$ 的盒子。当优化自由参数 $\boldsymbol{W}$ 与 $\boldsymbol{b}$ 时，输出的误差可以通过盒子返回，按照所选算法进行增量更新。需要注意的是，甚至还可以把误差传递给输入，计算输出的误差。对于线性模型，把误差传递给输入是不需要的，但是正如将在 5.4.3 节中看到的一样，对于反向传播算法而言，把误差传递给输入却是必要的。广义的线性模型如图 5-4 所示。

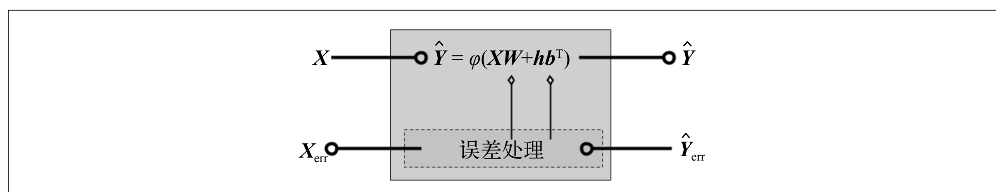


图 5-4：线性模型

然后可以实现 `LinearModel` 类，它只负责保存输出函数的类型、自由参数的状态，以及更新参数的简单方法。

```
public class LinearModel {

    private RealMatrix weight;
    private RealVector bias;
    private final OutputFunction outputFunction;

    public LinearModel(int inputDimension, int outputDimension,
    OutputFunction outputFunction) {
        weight = MatrixOperations.getUniformRandomMatrix(inputDimension,
        outputDimension, 0L);
        bias = MatrixOperations.getUniformRandomVector(outputDimension, 0L);
```

```
        this.outputFunction = outputFunction;
    }

    public RealMatrix getOutput(RealMatrix input) {
        return outputFunction.getOutput(input, weight, bias);
    }

    public void addUpdateToWeight(RealMatrix weightUpdate) {
        weight = weight.add(weightUpdate);
    }

    public void addUpdateToBias(RealVector biasUpdate) {
        bias = bias.add(biasUpdate);
    }
}
```

输出函数的接口如下：

```
public interface OutputFunction {
    RealMatrix getOutput(RealMatrix input, RealMatrix weight, RealVector bias);
    RealMatrix getDelta(RealMatrix error, RealMatrix output);
}
```

大多数情况下都无法精确地确定 $\boldsymbol{W}$ 与 $\boldsymbol{b}$，使得 $\boldsymbol{X}$ 与 $\boldsymbol{Y}$ 严格满足它们之间的关系。此处所能做的就是估计 $\boldsymbol{Y}$（称作 $\hat{\boldsymbol{Y}}$），然后使所选择的损失函数 $\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})$ 最小化。然后，目标是根据下式在一系列的迭代（记作 t）中增量更新 $\boldsymbol{W}$ 与 $\boldsymbol{b}$ 的值。

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t + \Delta\boldsymbol{W}_t$$
$$\boldsymbol{b}_{t+1} = \boldsymbol{b}_t + \Delta\boldsymbol{b}_t$$

本节集中讨论了使用梯度下降算法确定 $\boldsymbol{W}$ 与 $\boldsymbol{b}$ 的值。回想一下，损失函数终归是 $\boldsymbol{W}$ 与 $\boldsymbol{b}$ 的函数，可以使用梯度下降优化方法对损失函数的梯度进行增量更新。

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta\nabla\mathscr{L}(\boldsymbol{W})_t$$
$$\boldsymbol{b}_{t+1} = \boldsymbol{b}_t - \eta\nabla\mathscr{L}(\boldsymbol{b})_t$$

优化的目标函数是平均损失$\langle\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})\rangle$，其中，平均损失关于参数 w_{ij} 与 b_j 的梯度的各个项可以表示如下：

$$\frac{\partial\mathscr{L}}{\partial w} = \frac{\partial\mathscr{L}}{\partial\hat{y}}\frac{\partial\hat{y}}{\partial z}\frac{\partial z}{\partial w}$$

上式等号右边的第一项是损失函数的导数，在前一节中讨论过；上式等号右边的第二项是输出函数的导数。

$$\frac{\partial\hat{y}}{\partial z} = \varphi'(z)$$

第三项只是 z 关于 w 或者 b 的导数。

$$\frac{\partial z}{\partial w} = x$$

$$\frac{\partial z}{\partial b} = 1$$

正如我们所将看到的一样，选择适当的一对损失函数与输出函数，将带来数学上的简化，从而可以使用 delta 规则。在这种情况下，对权值以及偏移量的更新可以一直按下式进行：

$$\Delta \boldsymbol{W} = -\eta \boldsymbol{X}^{\mathrm{T}} (\boldsymbol{Y} - \boldsymbol{T})$$

$$\Delta \boldsymbol{b} = -\eta \boldsymbol{h}^{\mathrm{T}} (\boldsymbol{Y} - \boldsymbol{T})$$

当平均损失$\langle \mathscr{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) \rangle$在一定的数值容限范围内（例如 10E − 6）停止变化时，可以停止优化过程，且假定 $\boldsymbol{W}$ 与 $\boldsymbol{b}$ 已经到达其最优值。然而，由于一些奇异数值，迭代算法容易永远迭代下去。因此，所有迭代算法都要设置最大迭代次数（例如 1000），此后将终止迭代过程。总是检查是否达到了最大迭代次数是个好习惯，因为损失的变化可能仍然很大，这表明还没有找到自由参数的最优值。转换函数 $\varphi(\boldsymbol{Z})$ 与损失函数$\langle \mathscr{L}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) \rangle$的形式取决于所要解决的问题。下面讨论几种常见场景的细节。

1. 线性回归

对线性回归而言，$\varphi(\boldsymbol{Z})$ 设置为恒等函数，即输出与输入相等。

$$\varphi(\boldsymbol{Z}) = \boldsymbol{Z}$$

这就是我们熟悉的线性回归模型的形式：

$$\hat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{h}\boldsymbol{b}^{\mathrm{T}}$$

第 2 章与第 3 章中采用不同方法对这个问题进行了求解。第 2 章中把该问题用矩阵来表示，然后采用一个逆向求解器来求解自由参数。在第 3 章中则用最小二乘方法进行求解。然而，求解这个问题甚至还有许多其他方法。岭回归、lasso 回归以及弹性网络只是其中几个例子。指导思想是通过对变量参数的惩罚以消去在优化过程中没有用处的那些变量。

```
public class LinearOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
    RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias);
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // 输出梯度全部是1，……因此只返回errorGradient
        return errorGradient;
    }

}
```

2. logistic 回归

用于求解 y 是 0 或 1 的问题，也可以是多维问题，例如 $\boldsymbol{y} = 0, 1, 1, 0, 1$。

对于非线性函数 $\varphi(z)=\dfrac{1}{1+\exp(-z)}$，采用梯度下降法时，需要函数的导数：

$$\varphi'(z)=\frac{\exp(-z)}{(1+\exp(-z))^2}$$

注意，导数也可以方便地用原始函数来表示。这是因为它允许重用关于 φ 的计算结果，而不是必须对所有计算代价高昂的矩阵代数进行重新计算。

$$\varphi'(z)=\varphi(z)(1-\varphi(z))$$

那么，梯度下降法可以像下面这样实现：

```
public class LogisticOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
    RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias, new Sigmoid());
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // 这将永久改变output
        output.walkInOptimizedOrder(new UnivariateFunctionMapper(
        new LogisticGradient()));

        // 现在的output是原output的梯度
        return MatrixOperations.ebeMultiply(errorGradient, output);
    }

    private class LogisticGradient implements UnivariateFunction {

        @Override
        public double value(double x) {
            return x * (1 - x);
        }
    }
}
```

采用交叉熵损失函数计算损失项时，注意到对于 $\hat{y}=\varphi(z)$ 有下式：

$$\frac{\partial\mathscr{L}}{\partial\hat{y}}\frac{\partial\hat{y}}{\partial z}=\frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\hat{y}(1-\hat{y})$$

因此，接下来可以把它化简为下式：

$$\frac{\partial\mathscr{L}}{\partial z}=\hat{y}-y$$

再注意到有下面的等式：

$$\frac{\partial z}{\partial w} = x$$

损失函数相对于权值的梯度可以表示为下式：

$$\frac{\partial \mathscr{L}}{\partial w} = (\hat{y} - y)x$$

可以引入学习速率 η 来减缓更新过程。针对所使用的数据矩阵进行调整后，最终公式如下：

$$\Delta \boldsymbol{W} = -\eta \boldsymbol{X}^{\mathrm{T}} (\boldsymbol{Y} - \boldsymbol{T})$$
$$\Delta \boldsymbol{b} = -\eta \boldsymbol{h}^{\mathrm{T}} (\boldsymbol{Y} - \boldsymbol{T})$$

此处，$\boldsymbol{h}$ 是 m 维向量，其分量全部为 1。注意到引入学习速率 η，它通常取值在 0.0001~1 范围内，用于限制参数收敛的速度。对于较小的 η 值，更可能找到正确的权值，但代价是需要完成更多次耗时的迭代。对于较大的 η 值，将以快得多的速度完成算法的学习任务，然而可能会不经意地略过最优解，得到无意义的权值。

3. 归一化指数（softmax）回归

归一化指数（softmax）回归与 logistic 回归类似，但是目标变量可以是多元的（介于 0 与 `numClasses` − 1 的整数）。可以用一位有效编码对输出进行转换，使得 $\boldsymbol{Y} = \{0,0,1,0\}$。注意，与多输出的 logistic 回归不同，此处在每一行中只有一个位置能够设置为 1，而所有其他位置必须为 0。接着，对转换后的矩阵的每个元素求指数函数（exp）的值，然后再逐行进行 L1 归一化。

$$\varphi(z_{ij}) = \frac{\exp(z_{ij})}{\Sigma_j \exp(z_{ij})}$$

因为导数涉及多个变量，所以雅可比矩阵（Jacobian）用下面的项取代了梯度：

$$\varphi'(z) = \begin{cases} \varphi_i(z)(1 - \varphi_i(z)), & i = j \\ -\varphi_i(z)\varphi_j(z) \quad , & i \neq j \end{cases}$$

于是，对于单个的 p 维输出以及预测，可以计算下面的量：

$$\frac{\partial \mathscr{L}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial z} = \begin{pmatrix} \frac{-y_1}{\hat{y}_1} \frac{-y_2}{\hat{y}_2} \cdots \frac{-y_p}{\hat{y}_p} \end{pmatrix} \begin{pmatrix} \hat{y}_1(1 - \hat{y}_1) & -\hat{y}_1\hat{y}_2 & \cdots & -\hat{y}_1\hat{y}_p \\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1 - \hat{y}_2) & \cdots & -\hat{y}_2\hat{y}_p \\ \vdots & \vdots & \ddots & \vdots \\ -\hat{y}_p\hat{y}_1 & -\hat{y}_p\hat{y}_2 & \cdots & \hat{y}_p(1 - \hat{y}_p) \end{pmatrix}$$

可以简化为下式：

$$\frac{\partial \mathscr{L}}{\partial z} = ((\hat{y}_1 - y_1)(\hat{y}_2 - y_2)\cdots(\hat{y}_p - y_p))$$

每个项的更新规则与采用梯度下降法的线性模型完全相同。

$$\frac{\partial \mathscr{L}}{\partial w} = (\hat{y} - y)x$$

这里有个实际问题，那就是需要对输入进行两轮计算来得到归一化指数输出。首先，对每个参数求指数函数的值，并随时记录累加和。然后，再次对这个列表进行迭代，每个项都除以累加和。当（且仅当）采用归一化指数交叉熵作为误差时，系数的更新公式与 logistic 回归的更新公式完全相同。5.4.3 节中将显式地给出这个计算。

```
public class SoftmaxOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
                                RealVector bias) {
        RealMatrix output = MatrixOperations.XWplusB(input, weight, bias,
            new Exp());
        MatrixScaler.l1(output);
        return output;
    }

    @Override
    public RealMatrix getDelta(RealMatrix error, RealMatrix output) {

        RealMatrix delta = new BlockRealMatrix(error.getRowDimension(),
        error.getColumnDimension());

        for (int i = 0; i < output.getRowDimension(); i++) {
            delta.setRowVector(i, getJacobian(output.getRowVector(i)).
            preMultiply(error.getRowVector(i)));
        }
        return delta;
    }

    private RealMatrix getJacobian(RealVector output) {

        int numRows = output.getDimension();
        int numCols = output.getDimension();
        RealMatrix jacobian = new BlockRealMatrix(numRows, numCols);
        for (int i = 0; i < numRows; i++) {
            double output_i = output.getEntry(i);
            for (int j = i; j < numCols; j++) {
                double output_j = output.getEntry(j);
                if(i==j) {
                    jacobian.setEntry(i, i, output_i*(1-output_i));
                } else {
                    jacobian.setEntry(i, j, -1.0 * output_i * output_j);
                    jacobian.setEntry(j, i, -1.0 * output_j * output_i);
                }
```

```
            }
        }
        return jacobian;
    }
}
```

4. 双曲正切函数

另一种常见的激活函数利用了双曲正切函数 tanh(z)，如下式所示：

$$\varphi(z) = \tanh(z)$$
$$\varphi'(z) = 1 - \tanh^2(z) = 1 - \varphi(z)^2$$

再一次，导数 $\varphi'(z)$ 重用了从 $\varphi(z)$ 中计算的结果。

```
public class TanhOutputFunction implements OutputFunction {

    @Override
    public RealMatrix getOutput(RealMatrix input, RealMatrix weight,
                                RealVector bias) {
        return MatrixOperations.XWplusB(input, weight, bias, new Tanh());
    }

    @Override
    public RealMatrix getDelta(RealMatrix errorGradient, RealMatrix output) {
        // 这永久地修改了output
        output.walkInOptimizedOrder(
            new UnivariateFunctionMapper(new TanhGradient()));

        // output现在是原来output的梯度
        return MatrixOperations.ebeMultiply(errorGradient, output);
    }

    private class TanhGradient implements UnivariateFunction {
        @Override
        public double value(double x) {
            return (1 - x * x);
        }
    }
}
```

5. 线性模型估计器

利用梯度下降算法以及适当的损失函数，可以构造简单的线性估计器，它采用 delta 规则对参数进行迭代更新。这个仅适用于输出函数以及损失函数配对正确的情形，如表 5-1 所示。

表5-1：delta规则中的配对

输出函数	损失函数
线性	平方
logistic	伯努利交叉熵
归一化指数	多项交叉熵
双曲正切	两点交叉熵

然后，可以扩展 IterativeLearningProcess 类，并添加输出函数预测以及更新的代码。

```
public class LinearModelEstimator extends IterativeLearningProcess {

    private final LinearModel linearModel;
    private final Optimizer optimizer;

    public LinearModelEstimator(
            LinearModel linearModel,
            LossFunction lossFunction,
            Optimizer optimizer) {
        super(lossFunction);
        this.linearModel = linearModel;
        this.optimizer = optimizer;
    }

    @Override
    public RealMatrix predict(RealMatrix input) {
        return linearModel.getOutput(input);
    }

    @Override
    protected void update(RealMatrix input, RealMatrix target,
                          RealMatrix output) {
        RealMatrix weightGradient =
            input.transpose().multiply(output.subtract(target));
        RealMatrix weightUpdate = optimizer.getWeightUpdate(weightGradient);
        linearModel.addUpdateToWeight(weightUpdate);

        RealVector h = new ArrayRealVector(input.getRowDimension(), 1.0);
        RealVector biasGradient = output.subtract(target).preMultiply(h);
        RealVector biasUpdate = optimizer.getBiasUpdate(biasGradient);
        linearModel.addUpdateToBias(biasUpdate);
    }

    public LinearModel getLinearModel() {
        return linearModel;
    }

    public Optimizer getOptimizer() {
        return optimizer;
    }
}
```

6. Iris 示例

Iris 数据集是探究线性分类器很好的示例。

```
/* 获得数据并拆分为训练集与测试集 */
Iris iris = new Iris();
MatrixResampler resampler = new MatrixResampler(iris.getFeatures(),
iris.getLabels());
resampler.calculateTestTrainSplit(0.40, 0L);
```

```
/* 构造线性估计器 */
LinearModelEstimator estimator = new LinearModelEstimator(
    new LinearModel(4, 3, new SoftmaxOutputFunction()),
    new SoftMaxCrossEntropyLossFunction(),
    new DeltaRule(0.001));

estimator.setBatchSize(0);
estimator.setMaxIterations(6000);
estimator.setTolerance(10E-6);

/* 模型参数的学习 */
estimator.learn(resampler.getTrainingFeatures(), resampler.getTrainingLabels());

/* 根据测试数据进行预测 */
RealMatrix prediction = estimator.predict(resampler.getTestingFeatures());

/* 结果 */
ClassifierAccuracy accuracy = new ClassifierAccuracy(prediction,
    resampler.getTestingLabels());

estimator.isConverged();            // true
estimator.getNumIterations();       // 3094
estimator.getLoss();                // 0.0769
accuracy.getAccuracy();             // 0.983
accuracy.getAccuracyPerDimension(); // {1.0, 0.92, 1.0}
```

5.4.3 深度网络

把某个线性模型的输出提供给另一个线性模型，作为后者的输入，可以产生非线性系统，该系统可以模拟复杂的行为。有多层的系统称作深度网络。线性模型具有输入与输出，深度网络在输入与输出之间添加了多个“隐藏的”层。关于深度网络的大多数解释是把输入层、隐藏层以及输出层称作独立的量。然而，本书采用另外一种观点，即深度网络不过是由多个线性模型组成的。于是，可以把深度网络看作纯粹的线性代数问题。图 5-5 说明了多层神经网络是如何可以被看作一系列线性模型的。

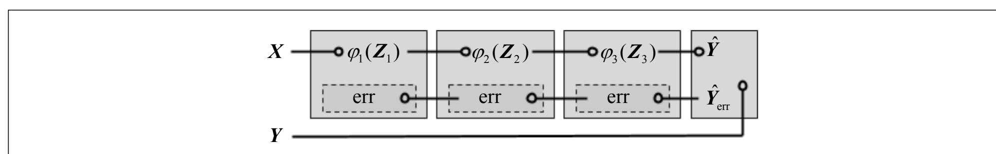


图 5-5：深度网络

1. 网络层

可以把线性模型的概念扩展为网络层的形式，在网络层中必须保留输入、输出和误差。于是，网络层的代码是对 `LinearModel` 类的扩展。

```
public class NetworkLayer extends LinearModel {

    RealMatrix input;
    RealMatrix inputError;
    RealMatrix output;
    RealMatrix outputError;
    Optimizer optimizer;

    public NetworkLayer(int inputDimension, int outputDimension,
            OutputFunction outputFunction, Optimizer optimizer) {
        super(inputDimension, outputDimension, outputFunction);
        this.optimizer = optimizer;
    }

    public void update() {

        // 反向传播误差
        /* D等于eps o f'(XW)，其中o是Hadamard积；
        或者等于Jf'(XW)，其中J是雅可比矩阵*/
        RealMatrix deltas = getOutputFunction().getDelta(outputError, output);

        /* E_out = DW^T */
        inputError = deltas.multiply(getWeight().transpose());

        /* W = W - alpha * delta * input */
        RealMatrix weightGradient = input.transpose().multiply(deltas);

        /* w_{t+1} = w_{t} + delta w_{t} */
        addUpdateToWeight(optimizer.getWeightUpdate(weightGradient));

        // 这本质上是对delta的列进行求和，该向量就是gradb
        RealVector h = new ArrayRealVector(input.getRowDimension(), 1.0);
        RealVector biasGradient = deltas.preMultiply(h);
        addUpdateToBias(optimizer.getBiasUpdate(biasGradient));
    }
}
```

2. 前馈

为了计算网络输出，必须正向通过网络每一层，以向各层输入提供数据。用网络输入 $\boldsymbol{X}_1$ 计算第一层的输出：

$$\hat{\boldsymbol{Y}}_1 = \varphi(\boldsymbol{X}_1\boldsymbol{W}_1 + \boldsymbol{h}\boldsymbol{b}_1^{\mathrm{T}})$$

设定第一层的输出作为第二层的输入：

$$\boldsymbol{X}_2 = \hat{\boldsymbol{Y}}_1$$

第二层的输出如下：

$$\hat{\boldsymbol{Y}}_2 = \varphi(\boldsymbol{X}_2\boldsymbol{W}_2 + \boldsymbol{h}\boldsymbol{b}_2^{\mathrm{T}}) = \varphi(\hat{\boldsymbol{Y}}_1\boldsymbol{W}_2 + \boldsymbol{h}\boldsymbol{b}_2^{\mathrm{T}})$$

通常，第一层之后每一层 l 的输出可以用前一层的输出表示如下：

$$\hat{\boldsymbol{Y}}_l = \varphi(\hat{\boldsymbol{Y}}_{l-1}\boldsymbol{W}_l + \boldsymbol{h}\boldsymbol{b}_l^{\mathrm{T}})$$

于是，L 层的前馈过程看上去像一系列嵌套的线性模型：

$$\hat{\boldsymbol{Y}}_L = \varphi_L(\cdots\varphi_2(\varphi_1(\boldsymbol{X}_1\boldsymbol{W}_1 + \boldsymbol{h}\boldsymbol{b}_1^{\mathrm{T}})\boldsymbol{W}_2 + \boldsymbol{h}\boldsymbol{b}_2^{\mathrm{T}})\cdots\boldsymbol{W}_L + \boldsymbol{h}\boldsymbol{b}_L^{\mathrm{T}})$$

有另一种更简单的方式，可以把这个表达式写作函数的结合：

$$\hat{\boldsymbol{Y}}_L = \varphi_L \circ \cdots \circ \varphi_2 \circ \varphi_1(\boldsymbol{Z})$$

除了独特的权值之外，每一层都可以采用不同形式的激活函数。这样，可以明确地知道前馈深度神经网络（也称作多层感知器）不过是由任意线性模型组成的而已。然而，结果却是非常复杂的非线性模型。

3. 反向传播

此时需要反向传播网络输出误差。对于最后一层（输出层），反向传播损失的梯度$\nabla\mathscr{L}(\boldsymbol{Y},\hat{\boldsymbol{Y}})$。对于兼容的损失函数 − 输出函数对，与线性模型估计器相同。就像 5.1.2 节中所陈述的那样，定义新量是便利的，这个量就是 delta 层，$\boldsymbol{D}$，它是 $\boldsymbol{Y}_{\mathrm{err}}$ 与输出函数的雅可比矩阵张量的矩阵乘积：

$$\boldsymbol{D} = \hat{\boldsymbol{Y}}_{\mathrm{err}}\boldsymbol{J}_{\varphi}^{(m)}$$

在大多数情况下，采用输出函数的梯度就足够了，上面的表达式可以简化为下式：

$$\boldsymbol{D} = \hat{\boldsymbol{Y}}_{\mathrm{err}} \circ \varphi'(\boldsymbol{Z})$$

此处要存储 $\boldsymbol{D}$，因为有两处要用到它，且必须按照顺序来计算。首先对反向传播误差进行更新：

$$\boldsymbol{X}_{\mathrm{err}} = \boldsymbol{D}\boldsymbol{W}^{\mathrm{T}}$$

接下来，按下式计算权值的梯度以及偏移量的梯度，其中 $\boldsymbol{h}$ 是分量全部为 1 且维数为 m 的向量：

$$\nabla\boldsymbol{W} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{D}$$
$$\nabla\boldsymbol{b} = \boldsymbol{h}^{\mathrm{T}}\boldsymbol{D}$$

注意表达式 $\boldsymbol{h}^{\mathrm{T}}\boldsymbol{D}$ 等价于对 $\boldsymbol{D}$ 的每一列进行求和。然后，各层的权值可以采用所选择的优化规则（通常是梯度下降法的某种变形）进行更新。这可以完成网络层所需的所有计算！然后设置下面一层的 $\boldsymbol{Y}_{\mathrm{err}}$ 为新计算得到的 $\boldsymbol{X}_{\mathrm{err}}$，并重复这个过程，直至第一层的参数被更新。



确保在更新权值之前先计算反向传播误差!

4. 深度网络估计器

采用与线性模型相同的迭代过程，可以学习深度网络的参数。在这种情况下，完整的前馈过程作为一个预测步骤，反向传播过程作为一个更新步骤。通过扩展 IterativeLearningProcess 类，并采用子类 NetworkLayer 构造线性模型的各个层，从而实现了深度网络估计器。

```
public class DeepNetwork extends IterativeLearningProcess {

    private final List<NetworkLayer> layers;

    public DeepNetwork() {
        this.layers = new ArrayList<>();
    }

    public void addLayer(NetworkLayer networkLayer) {
        layers.add(networkLayer);
    }

    @Override
    public RealMatrix predict(RealMatrix input) {

        /* 初始输入input必须被深度复制，否则会被覆盖 */
        RealMatrix layerInput = input.copy();

        for (NetworkLayer layer : layers) {
            layer.setInput(layerInput);

            /* 计算输出output，并且设置为下一层的输入input */
            RealMatrix output = layer.getOutput(layerInput);
            layer.setOutput(output);

            /*
                不需要进行深度复制，但是要注意每一层的输入input，
                与前一层的输出output共享内存
            */
            layerInput = output;
        }
        /* layerInput拥有最终的输出output……进行深度复制 */
        return layerInput.copy();
    }

    @Override
    protected void update(RealMatrix input, RealMatrix target,
                          RealMatrix output) {

        /* 获得网络误差的梯度，启动反向传播过程 */
        RealMatrix layerError = getLossFunction()
                                   .getLossGradient(output, target).copy();
```

```
        /* 创建列表的迭代器，并设定游标指向最后一层 */
        ListIterator li = layers.listIterator(layers.size());

        while (li.hasPrevious()) {
            NetworkLayer layer = (NetworkLayer) li.previous();
            /* 从更高层获得误差输入 */
            layer.setOutputError(layerError);
            /* 反向传播误差，并更新权值 */
            layer.update();
            /* 向下一层传递误差 */
            layerError = layer.getInputError();
        }
    }
}
```

5. MNIST 示例

MNIST 是经典的手写数字数据集，常用于测试学习算法。此处通过使用拥有两个隐藏层的简单网络获得了 94% 的准确率。

```
MNIST mnist = new MNIST();

DeepNetwork network = new DeepNetwork();

/* 输入层、隐藏层以及输出层 */
network.addLayer(new NetworkLayer(784, 500, new TanhOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

network.addLayer(new NetworkLayer(500, 300, new TanhOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

network.addLayer(new NetworkLayer(300, 10, new SoftmaxOutputFunction(),
    new GradientDescentMomentum(0.0001, 0.95)));

/* 运行时参数 */
network.setLossFunction(new SoftMaxCrossEntropyLossFunction());
network.setMaxIterations(6000);
network.setTolerance(10E-9);
network.setBatchSize(100);

/* 学习 */
network.learn(mnist.trainingData, mnist.trainingLabels);

/* 预测 */
RealMatrix prediction = network.predict(mnist.testingData);

/* 计算准确率 */
ClassifierAccuracy accuracy =
    new ClassifierAccuracy(prediction, mnist.testingLabels);

/* 结果 */
network.isConverged(); // false
network.getNumIterations(); // 10000
network.getLoss(); // 0.00633
accuracy.getAccuracy(); // 0.94
```

第6章

Hadoop MapReduce

如果需要底层控制，并且想要优化大数据管道或者使其效率更高，那么需要用 Java 编写 MapReduce 任务。采用 MapReduce 并不是必需的，却是值得的，因为它是一个设计良好的系统以及 API。学习关于 MapReduce 的基本知识可以让你走得更快、更远，不过在着手编写定制的 MapReduce 任务之前，不要忽略了诸如 Apache Drill 之类的工具，它们可以在 Hadoop 上编写出标准的 SQL 查询语句。

本章假设在本地计算机上或者通过访问 Hadoop 集群，可以运行 Hadoop 分布式文件系统（HDFS）。为了模拟实际 MapReduce 任务是如何运行的，可以在同一个节点上以伪分布方式（pseudodistributed mode）运行 Hadoop，这个节点可以是本地节点，也可以是远程节点。考虑到如今在机箱（或者便携式计算机）中能够安装多少 CPU、RAM 以及存储资源这一事实，我们本质上可以构造微型超级计算机，它能够运行相当大的分布式任务。可以在本地计算机上（针对数据子集）进行处理，并且对应用进行调试，在调试好应用后再将其扩展至整个集群。

如果合理地安装了 Hadoop 的客户端，那么只需键入下列命令，就可以得到所有可用的 Hadoop 操作的完整列表：

```
bash$ hadoop
```

6.1 Hadoop分布式文件系统

Apache Hadoop 有命令行工具，用来访问 Hadoop 的文件系统，并启动 MapReduce 任务。采用下面的方式可以调用文件系统访问命令 fs：

```
bash$ hadoop fs <command> <args>
```

该命令是由前导连字符以及任意数目的标准 UNIX 文件系统命令组成的，例如 `ls`、`cd` 或 `mkdir`。例如，为了列出 HDFS 根目录中的所有项，可以键入下述命令：

```
bash$ hadoop fs -ls /
```

注意，访问根目录时要加上斜杠 /。如果没有加上斜杠，那么上述命令什么结果也不会返回，这样会让你误以为自己的 HDFS 是空的。键入命令 **hadoop fs** 将打印出所有可用的文件系统操作。一些更有用的操作包括：把数据复制到 HDFS，或者把数据从 HDFS 复制到其他地方；删除目录；在目录中合并数据。

把本地文件复制到 Hadoop 文件系统中的命令如下：

```
bash$ hadoop fs -copyFromLocal <localSrc> <dest>
```

把文件从 HDFS 复制到本地驱动器的命令如下：

```
bash$ hadoop fs -copyToLocal <hdfsSrc> <localDest>
```

运行 MapReduce 任务之后，在任务的输出目录中很可能有许多文件。用户不需要逐个获取这些文件，Hadoop 有个便利的操作，可以把这些文件合并为一个文件，然后把结果存储在本地。

```
bash$ hadoop fs -getmerge <hdfs_output_dir> <my_local_dir>
```

如果 MapReduce 检测到输出目录已经存在，则几乎会马上导致 MapReduce 任务失败。因此，运行 MapReduce 任务的一项必要操作是，如果输出目录已经存在，则首先要移除该目录。

```
bash$ hadoop fs -rm rf <hdfs_dir>
```

6.2 MapReduce体系结构

MapReduce 运行分布式计算中**令人头疼**的**并行**范例。最初，把数据拆分为多个分块，再把这些分块送至相同的映射器（mapper）类，从数据中逐行提取键－值对。接下来再把这些键－值对划分为键－列表对，其中列表是经过排序的。通常，划分后的键－列表对个数就是归约任务的个数，但是并非必须如此。事实上，多个键－列表对可以在相同的分区中，且由相同的归约器（reducer）处理，但是可以确保每个键－列表对不会被分割为跨分区或者由不同的归约器进行处理。图 6-1 中显示了数据在 MapReduce 框架中传递的一般流程。

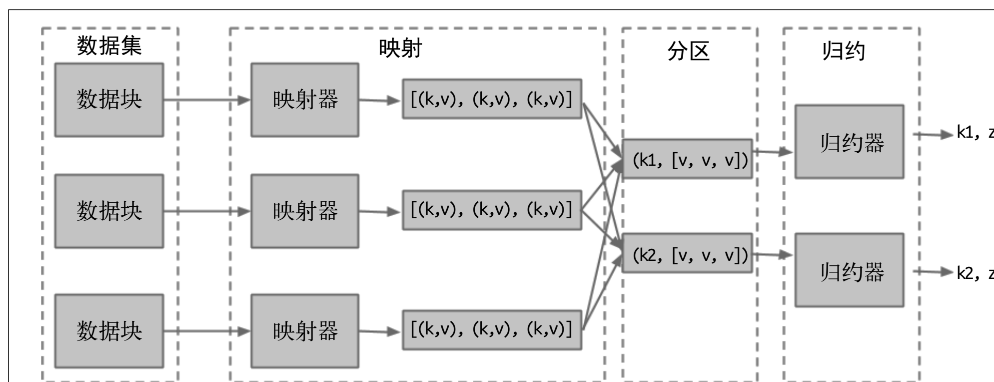


图 6-1：MapReduce 架构

例如，假设有如下的数据：

```
San Francisco, 2012
New York, 2012
San Francisco, 2017
New York, 2015
New York, 2016
```

对于数据集中的每一行，映射器可能输出像（`San Francisco, 2012`）这样的键 – 值对。然后分区器（partitioner）会按照键收集数据并对列表中的值进行排列。

```
(San Francisco, [2012, 2017])
(New York, [2012, 2015, 2016])
```

可以设定归约器的功能为输出最大的年份，使得（写入到输出目录中的）最终结果如下：

```
San Francisco, 2017
New York, 2016
```

Hadoop MapReduce API 允许使用复合键和定制比较器对键进行分区以及对值排序，考虑到这一点很重要。

6.3 编写MapReduce应用

在 Hadoop 生态系统中，尽管有多种存储以及移动数据的方法，但是本章将集中讨论纯文本文件。不论底层数据是用字符串、CSV、TSV 还是 JSON 数据字符串的形式存储的，都可以轻松地对数据进行读取、共享以及处理。Hadoop 也提供了能够读写自己的 Sequence 与 Map 文件格式的资源，有时可能想要探究各种第三方的序列化格式，例如 Apache Avro、Apache Thrift、Google Protobuf、Apache Parquet 以及其他格式。虽然所有这些格式都提供了操作方面以及效率方面的优势，但也必须要考虑到它们确实增加了复杂性。

6.3.1　剖析MapReduce任务

基本的 MapReduce 任务只有少数必要的功能。重写 `run()` 方法的实质就是在该方法中包含 `Job` 类的单例。

```
public class BasicMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new BasicMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {

        Job job = Job.getInstance(getConf());
        job.setJarByClass(BasicMapReduceExample.class);
        job.setJobName("BasicMapReduceExample");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return job.waitForCompletion(true) ? 0 : 1;
    }

}
```

注意，由于还没有定义任何 `Mapper` 类或 `Reducer` 类，以上任务会使用默认的类，该类把输入文件不加修改地复制到输出目录中。在深入研究如何定制 `Mapper` 类以及 `Reducer` 类之前，必须首先理解 Hadoop MapReduce 所需要的专有数据类型。

6.3.2　Hadoop数据类型

通过 MapReduce 应用传递数据时，必须采用既可靠又有效的格式。遗憾的是（根据 Hadoop 的作者），原生 Java 基本类型（例如 `boolean`、`int`、`double`）以及更复杂的类型（例如 `String`、`Map`）并不能很好地传递。由于这个原因，Hadoop 生态系统有自己独特的序列化类型，在所有 MapReduce 应用中，这些类型是必需的。注意，所有常规的 Java 类型在本书的 MapReduce 代码内部可以很好地工作。只有不同 MapReduce 组件（映射器与归约器）进行连接时，才把原生 Java 基本类型转换为 Hadoop 类型。

1. Writable 类型

在 Hadoop 中，原生 Java 基本类型都有相应的表示，但是最有用的类型是 `BooleanWritable`、`IntWritable`、`LongWritable` 以及 `DoubleWritable`。采用 `Text` 类型表示 Java 的 `String` 类型。采用 `NullWritable` 类型表示 `null`，在 MapReduce 任务中，若没有数据通过特别的键或值传递，则这种类型会派上用场。此外，当采用与 `userid` 或者其他独特标识符对应的散列值作为键时，甚至会用到 `MD5Hash` 类型。还有 `MapWritable` 类型，它用于创建可比较的

HashMap 版本的 Writable 类型。所有这些类型都是可比较的（例如，它们有 hash() 方法以及 equals() 方法，能够对 MapReduce 任务中的事件进行比较与排序）。当然，还有更多的类型，但是上述这些是其中一些比较有用的类型。一般来说，Hadoop 类型在构造方法中采用原生 Java 基本类型作为参数。

```
Int count = 42;
IntWritable countWritable = new IntWritable(count);

String data = "The is a test string";
Text text = new Text(data);
```

注意，在 Mapper 类以及 Reducer 类的代码内部使用原生 Java 基本类型。**只有**这些类的实例的键、值以及输入与输出必须使用 Hadoop 可写类型（如果是键，则还要求是可比较的），因为这是在 MapReduce 组件之间进行数据传递。

2. 定制的 Writable 类型与 WritableComparable 类型

有时需要使用 Hadoop 中没有包括的特殊类型。一般来说，Hadoop 类型必须实现 Writable 接口，它采用 write() 方法进行对象的序列化，采用 read() 方法进行对象的反序列化。然而，如果把对象用作键，那么它必须实现 WritableComparable 接口，因为在分区与排序时需要用到 compareTo() 方法以及 hashCode() 方法。

❑ Writable 类型

因为 Writable 接口只有两个方法，write() 以及 readFields()，所以基本的定制可写类型只需要重写这两个方法。然而，可以增加带参数的构造方法，使得可以像前面的例子中创建 IntWritable 以及 Text 实例那样来实例化对象。此外，若添加静态 read() 方法，则需要无参数的构造方法。

```
public class CustomWritable implements Writable {

    private int id;
    private long timestamp;

    public CustomWritable() {
    }

    public CustomWritable(int id, long timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    public void write(DataOutput out) throws IOException {
        out.writeInt(id);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
```

```
        timestamp = in.readLong();
    }

    public static CustomWritable read(DataInput in) throws IOException {
        CustomWritable w = new CustomWritable();
        w.readFields(in);
        return w;
    }
}
```

❑ WritableComparable 类型

如果把定制可写类型用作键，那么除了 write() 方法与 readField() 方法之外，还需要实现 hashCode() 方法与 compareTo() 方法。

```
public class CustomWritableComparable implements WritableComparable {

    private int id;
    private long timestamp;

    public CustomWritable() {
    }

    public CustomWritable(int id, long timestamp) {
        this.id = id;
        this.timestamp = timestamp;
    }

    public void write(DataOutput out) throws IOException {
        out.writeInt(id);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        id = in.readInt();
        timestamp = in.readLong();
    }

    public int compareTo(CustomWritableComparable o) {
        int thisValue = this.value;
        int thatValue = o.value;
        return (thisValue < thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
    }

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + (int) (timestamp ^ (timestamp >>> 32));
        return result
    }
}
```

6.3.3 映射器

Mapper 类把原始输入数据映射到通常较小的新数据结构。一般来说，并不需要输入文件中每一行的所有数据项，而仅是选择少数几个项。在某些情况下会完整地丢弃某些行。此时，需要决定让哪些数据进入下一轮的处理。可以把这一步看作对原始数据进行转换且过滤，剩下的就是确实需要的数据。如果在 MapReduce 任务中没有包括 Mapper 实例，那么将使用 IdentityMapper，它仅仅是把数据直接传递给归约器而已。如果连归约器也没有，那么本质上就是把输入数据复制到输出。

1. 通用映射器

在 Hadoop 中已经包括了几种常见的映射器，可以在 MapReduce 任务中指定。默认的映射器是 IdentityMapper，它输出的就是所输入的数据。InverseMapper 会把键与值的角色进行互换。还有 TokenCounterMapper，它把每个标记及其计数作为（Text， IntWritable）形式的键 - 值对进行输出。RegexMapper 以键以及常量值 1 的形式输出一个正则表达式匹配。如果这些都不适合于自己的应用，则可以考虑编写为自己定制的映射器实例。

2. 定制映射器

Mapper 类解析文本文件的方式与第 1 章中解析常规文本文件的行大致相同。所需要的仅是 map() 方法而已。map() 方法的基本目的就是解析输入的一行，并且通过 context.write() 方法输出键 - 值对。

```
public class ProductMapper extends
    Mapper<LongWritable, Text, IntWritable, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        try {

            /* 文件的每一行都是<userID>、<productID>、<timestamp> */
            String[] items = value.toString().split(",");
            int userID = Integer.parseInt(items[0]);
            String productID = items[1];
            context.write(new IntWritable(userID), new Text(productID));

        } catch (NumberFormatException | IOException | InterruptedException e) {

            context.getCounter("mapperErrors", e.getMessage()).increment(1L);
        }
    }

}
```

还有 setup() 方法与 cleanup() 方法。对 Mapper 类实例化时，就会运行一次 setup() 方法。也许你不需要它，但是它迟早会派上用场，例如每次调用 map() 方法时都会用到某个数据结构。同样，你可能不需要 cleanup() 方法，但是最后一次调用 map() 方法之后，就会调

用一次 cleanup() 方法，以进行一些清理操作。还有 run() 方法，它实际上做的是映射数据。没有真正的理由来重写 run() 方法，除非你有充足的理由实现自己的 run() 方法，否则最好不要修改它。在 6.4 节中将展示如何利用 setup() 方法做一些独特的计算。

为了使用定制的映射器，必须在 MapReduce 应用中指定它，并设置映射所输出的键与值的类型。

```
job.setMapperClass(ProductMapper.class);
job.setMapOutputKeyClass(IntWritable.class);
job.setMapOutputValueClass(Text.class);
```

6.3.4 归约器

Reducer 的作用是对与键对应的值的列表进行迭代，并计算出单一的输出值。当然，可以定制 Reducer 的输出类型，只要这个类型实现了 Writable，就可以返回所希望的任何类型。务必要注意，每个归约器都会处理至少一个键以及它所对应的所有值，因此不必担心属于某个键的某些值会被送到其他归约器进行处理。归约器的个数也就是输出文件的个数。

1. 通用归约器

如果没有指定 Reducer 实例，那么 MapReduce 任务直接把映射的数据发送给输出。在 Hadoop 库中有一些有用的归约器，迟早可以派上用场。IntSumReducer 类以及 LongSumReducer 类在各自的 reduce() 方法中，分别采用 IntWritable 类型以及 LongWritable 类型的整数作为值。输出就是所有值的累加和。对于 MapReduce 而言，由于计数十分常用，因此这些类十分便利。

2. 定制归约器

归约器的代码结构与映射器类似。通常只需要用自己的代码重写 reduce() 方法即可。有时，当构造所有归约器都必须用到的特定数据结构或者基于文件的资源时，会用到 setup() 方法。注意，因为在映射器阶段之后，特定键关联的所有数据会被分组并进行排序，放到 Iterable 类型的列表中，并作为归约器的输入，所以归约器的签名要采用 Iterable 类型的值。

```
public class CustomReducer extends
    Reducer<IntWritable, Text, IntWritable, IntWritable>{

    @Override
    protected void reduce(IntWritable key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException {

        int someValue = 0;

        /* 对值进行迭代，并进行相应的操作  */
        for (Text value : values) {
            //用value使someValue增大
        }
```

```
        context.write(key, new IntWritable(someValue));
    }

}
```

在 MapReduce 任务中，需要指定 Reducer 类及其键与值的输出类型。

```
job.setReducerClass(CustomReducer.class);
job.setOutputKeyClass(IntWritable.class);
job.setOutputValueClass(IntWritable.class);
```

6.3.5 JSON字符串作为文本的简单性

JSON 数据便于人们阅读，它的内置模式确实有用，许多工具可以接受 JSON 数据。由于这些原因，到处都可以看到 JSON 数据（其文件的每一行是独立的 JSON 字符串）。因为 Hadoop 的 Text 类型可以对 JSON 字符串序列化，所以在 MapReduce 应用中采用 JSON 数据作为输入，不再需要定制可写对象。这个过程可以像在 map() 方法中正确地使用 JSONObject 那么简单。也可以创建类，使用 value.toString() 方法的值来构造更加复杂的映射模式。

```
public class JSONMapper extends Mapper<LongWritable, Text, Text, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        JSONParser parser = new JSONParser();
        try {
            JSONObject obj = (JSONObject) parser.parse(value.toString());

            // 从这个对象中得到所需要的数据
            String userID = obj.get("user_id").toString();
            String productID = obj.get("product_id").toString();
            int numUnits = Integer.parseInt(obj.get("num_units").toString());

            JSONObject output = new JSONObject();
            output.put("productID", productID);
            output.put("numUnits", numUnits);

            /* 此处可以添加更多的键-值对，包括数组 */

            context.write(new Text(userID), new Text(output.toString()));


        } catch (ParseException ex) {
            // 解析JSON数据出错
        }

    }
}
```

把最终归约器得到的输出数据作为 `Text` 对象，效果也很好。为了后续数据管道能够有效地使用，最终数据文件采用 JSON 数据格式。现在归约器能够输入键 – 值对（`Text`，`Text`），可以采用 `JSONObject` 处理 JSON 数据。这种做法的好处是，不必再为这种数据结构创建复杂的定制 `WritableComparable` 类型。

6.3.6 部署技巧

运行 MapReduce 任务时，有许多选项以及命令行开关。记住，在运行任务之前，需要先删除输出目录。

```
bash$ hadoop fs -rm -r <path>/output
```

1. 运行独立程序

有时一定会遇到（可能是自己编写的）单个文件中包含了整个 MapReduce 任务的代码。唯一真正的区别在于，必须把任何 `Mapper`、`Reducer`、`Writable` 以及其他定制对象定义为静态的。除此之外，任务的机制完全相同。这种做法的显著优点是，有完全自包含的任务，不需要担心文件之间的依赖关系，因此不必担心 JAR 包以及其他问题。现在（在命令行中用 **`javac`** 命令）构建 Java 文件，像下面这样运行构建得到的类：

```
bash$ hadoop BasicMapReduceExample input output
```

2. 部署 JAR 应用

如果 MapReduce 任务属于更大的项目，该项目已经生成了 JAR 包，那么该 JAR 包可能包含多个这样的任务，需要根据 JAR 包进行部署，并为任务指定完整的 URI。

```
hadoop jar MyApp.jar com.datascience.BasicMapReduceExample input output
```

3. 说明依赖关系

在 MapReduce 任务中，必须像下面这样引入逗号分隔的文件列表：

```
-files file.dat, otherFile.txt, myDat.json
```

可以用逗号分隔的列表添加所需要的任何 JAR 包：

```
-libjars myJar.jar, yourJar.jar, math.jar
```

注意，诸如 **`-files`** 以及 **`-libjars`** 的命令行开关，必须放置在诸如输入 / 输出等任何命令参数之前。

4. 采用 bash 脚本进行简化

在某些时候，在命令行中输入所有这些文本是容易出错且费时费力的。为了查找上个星期启动的命令而查看 bash 历史时也是这样。可以为特定任务创建定制的脚本，该脚本中包括命令行参数，如输入 / 输出目录，甚至可以指定要运行哪个类。像下面这样，把它们全部

放入可执行 bash 脚本中：

```
#!/bin/bash

# 处理来自命令行的输入输出目录
INPUT=$1
OUTPUT=$2

# 在这个bash脚本中，下面几行是硬编码的
LIBJARS=/opt/math3.jar, morejars.jar
FILES=/usr/local/share/meta-data.csv, morefiles.txt
APP_JAR=/usr/local/share/myApp.jar
APP_CLASS=com.myPackage.MyMapReduceJob

# 删除输出目录
hadoop fs -rm -r $OUTPUT

# 启动任务
hadoop jar $APP_JAR $APP_CLASS -files $FILES -libjars $LIBJARS $INPUT $OUTPUT
```

然后，必须记住要让脚本成为可执行的（下面的命令只执行一次）：

```
bash$ chmod +x runMapReduceJob.sh
```

接下来像下面这样运行命令：

```
bash$ myJobs/runMapReduceJob.sh inputDirGoesHere outputDirGoesHere
```

如果从与脚本相同的目录运行任务，则可以用下面的命令：

```
bash$ ./runMapReduceJob.sh inputDirGoesHere outputDirGoesHere
```

6.4 MapReduce示例

为了真正掌握 MapReduce，需要进行实践。要想理解 MapReduce 如何工作，没有比立即着手解决问题更好的办法。尽管乍看上去，系统可能显得复杂且烦琐，但随着取得某些成功之后，会感受到系统的优点。此处有一些典型示例以及一些有代表性的计算。

6.4.1 单词计数

此处使用内置的映射器类 `TokenCounterMapper` 对标记计数，并采用内置的归约器类 `IntSumReducer` 对整数进行累加。

```
public class WordCountMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new WordCountMapReduceExample(), args);
        System.exit(exitCode);
```

```
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(WordCountMapReduceExample.class);
        job.setJobName("WordCountMapReduceExample");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(TokenCounterMapper.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setNumReduceTasks(1);

        return job.waitForCompletion(true) ? 0 : 1;
    }
}
```

该任务可以在含有任意类型文本文件的输入目录上运行。

```
hadoop jar MyApp.jar \\
com.datascience.WordCountMapReduceExample input output
```

可以用下面的命令查看输出结果：

```
hadoop fs -cat output/part-r-00000
```

6.4.2 定制单词计数

你可能已经注意到，内置的 TokenCounterMapper 类并没有产生希望的结果。因此，可以一直使用第 4 章的 SimpleTokenizer 类。

```
public class SimpleTokenMapper extends
    Mapper<LongWritable, Text, Text, LongWritable> {

    SimpleTokenizer tokenizer;

    @Override
    protected void setup(Context context) throws IOException {
        // 只保留多于3个字符的那些单词
        tokenizer = new SimpleTokenizer(3);
    }

    @Override
    protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
```

```
        String[] tokens = tokenizer.getTokens(value.toString());
        for (String token : tokens) {
            context.write(new Text(token), new LongWritable(1L));
        }

    }
}
```

一定要在任务中进行适当修改。

```
/* 设置映射器 */
job.setMapperClass(SimpleTokenMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

/* 设置归约器 */
job.setReducerClass(LongSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
```

6.4.3 稀疏线性代数

假设有大矩阵（稠密或者稀疏），在文件的每一行以 `<i,j,value>` 的格式存储矩阵元素所位于的行列编号（`i,j`）以及相应的元素值。该矩阵如此之大，以至于无法把它装载到 RAM 中以供进一步的线性代数例程使用。此处的目标是用提供的输入向量进行矩阵与向量相乘。由于向量已经被序列化，因此文件可以包含在 MapReduce 任务内。

假设已经把用逗号或者制表符分隔的文本文件存储在分布式文件系统中的多个节点上。如果数据在字面上以 `i,j,value` 格式的文本字符串（例如 34 290、1.2362）存储在文件的每一行，则可以编写简单的映射器来解析每一行。在这种情况下，可以进行矩阵相乘。你可能记得，矩阵相乘需要把矩阵的每一行乘上与之维数相同的列向量。输出向量的每个位置 `i` 与对应矩阵行的索引相同。因此，可以采用矩阵行索引 `i` 作为键。此处将创建定制可写类 `SparseMatrixWritable`，它包含行索引、列索引以及矩阵中每个元素的值。

```
public class SparseMatrixWritable implements Writable {
    int rowIndex; // i
    int columnIndex; // j
    double entry; // 在i,j位置的值

    public SparseMatrixWritable() {
    }

    public SparseMatrixWritable(int rowIndex, int columnIndex, double entry) {
        this.rowIndex = rowIndex;
        this.columnIndex = columnIndex;
        this.entry = entry;
    }

    @Override
```

```
    public void write(DataOutput d) throws IOException {
        d.writeInt(rowIndex);
        d.writeInt(rowIndex);
        d.writeDouble(entry);
    }

    @Override
    public void readFields(DataInput di) throws IOException {
        rowIndex = di.readInt();
        columnIndex = di.readInt();
        entry = di.readDouble();
    }

}
```

定制的映射器将读取文本的每一行，然后解析出这 3 个值，采用行索引作为键，采用 SparseMatrixWritable 作为值。

```
public class SparseMatrixMultiplicationMapper
extends Mapper<LongWritable, Text, IntWritable, SparseMatrixWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        try {
            String[] items = value.toString().split(",");
            int rowIndex = Integer.parseInt(items[0]);
            int columnIndex = Integer.parseInt(items[1]);
            double entry = Double.parseDouble(items[2]);
            SparseMatrixWritable smw = new SparseMatrixWritable(
            rowIndex, columnIndex, entry);
            context.write(new IntWritable(rowIndex), smw);
            // 注意：可以添加另一个context.write()。例如，若矩阵是稀疏的上三角阵时，
            // 则可以将context.write()用于对称的矩阵元素
        } catch (NumberFormatException | IOException | InterruptedException e) {
            context.getCounter("mapperErrors", e.getMessage()).increment(1L);
        }
    }
}
```

归约器必须在 setup() 方法中加载输入向量，在 reduce() 方法中从 SparseMatrixWritable 的列表中提取列索引，并把它们添加到稀疏向量中。计算输入向量与稀疏向量的点积，可以得到该键的输出值（例如那个索引对应的结果向量的值）。

```
public class SparseMatrixMultiplicationReducer extends Reducer<IntWritable,
                       SparseMatrixWritable, IntWritable, DoubleWritable>{

    private RealVector vector;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {
```

```
        /* 对RealVector对象进行反序列化 */

        // 注意这仅是文件名，在运行时请通过-files在分布式缓存中引入资源本身
        // 用set("vectorFileName", "此处是实际的文件名")在任务配置文件中设置文件名

        String vectorFileName = context.getConfiguration().get("vectorFileName");
        try (ObjectInputStream in = new ObjectInputStream(
        new FileInputStream(vectorFileName))) {
            vector = (RealVector) in.readObject();
        } catch(ClassNotFoundException e) {
            // 错误处理
        }
    }

    @Override
    protected void reduce(IntWritable key, Iterable<SparseMatrixWritable> values,
    Context context)
        throws IOException, InterruptedException {

        /* 基于rowVector维度与输入向量维度相等这一事实 */
        RealVector rowVector = new OpenMapRealVector(vector.getDimension());

        for (SparseMatrixWritable value : values) {
            rowVector.setEntry(value.columnIndex, value.entry);
        }

        double dotProduct = rowVector.dotProduct(vector);

        /* 因为矩阵与向量的乘积可能是稀疏的，所以只写非零的输出 */
        if(dotProduct != 0.0) {
            /* 此处输出向量索引及其值 */
            context.write(key, new DoubleWritable(dotProduct));
        }
    }
}
```

任务可以按如下设置后运行：

```
public class SparseAlgebraMapReduceExample extends Configured implements Tool {

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new SparseAlgebraMapReduceExample(), args);
        System.exit(exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(SparseAlgebraMapReduceExample.class);
        job.setJobName("SparseAlgebraMapReduceExample");

        // 第三个命令行参数是向量序列化后所形成文件的路径
        job.getConfiguration().set("vectorFileName", args[2]);
```

```
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(SparseMatrixMultiplicationMapper.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(SparseMatrixWritable.class);
        job.setReducerClass(SparseMatrixMultiplicationReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(DoubleWritable.class);
        job.setNumReduceTasks(1);

        return job.waitForCompletion(true) ? 0 : 1;
    }
}
```

设置好任务后，可以采用下面的命令运行：

```
hadoop jar MyApp.jar \\
com.datascience.SparseAlgebraMapReduceExample \\
-files /<path>/RandomVector.ser input output RandomVector.ser
```

可以用下面的命令查看输出结果：

```
hadoop fs -cat output/part-r-00000
```

附录A

数据集

本书所有数据集都保存在 src/main/resources/datasets 路径下，Java 类代码存放在 src/main/java 路径下，用户资源保存在 src/main/resources 路径下。一般来说，采用 JAR 的加载功能可以直接从 JAR 包中获取文件的内容，而不是从文件系统中获取。

A.1 Anscombe的4组数据

Anscombe 的 4 组数据（Anscombe's quartet）是关于 4 组 *x-y* 对的数据集合，它具有一些值得注意的特性。尽管 4 组数据绘制的图形完全不同，但是它们具有使得统计结果几乎相同的特性。4 组数据的值如表 A-1 所示。

表A-1：Anscombe的4组数据

x1	y1	x2	y2	x3	y3	x4	y4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

可以轻松地把数据作为静态成员硬编码到类中：

```
public class Anscombe {
    public static final double[] x1 = {10.0, 8.0, 13.0, 9.0, 11.0,
                                       14.0, 6.0, 4.0, 12.0, 7.0, 5.0};
    public static final double[] y1 = {8.04, 6.95, 7.58, 8.81, 8.33,
                                       9.96, 7.24, 4.26, 10.84, 4.82, 5.68};
    public static final double[] x2 = {10.0, 8.0, 13.0, 9.0, 11.0,
                                       14.0, 6.0, 4.0, 12.0, 7.0, 5.0};
    public static final double[] y2 = {9.14, 8.14, 8.74, 8.77, 9.26,
                                       8.10, 6.13, 3.10, 9.13, 7.26, 4.74};
    public static final double[] x3 = {10.0, 8.0, 13.0, 9.0, 11.0,
                                       14.0, 6.0, 4.0, 12.0, 7.0, 5.0};
    public static final double[] y3 = {7.46, 6.77, 12.74, 7.11, 7.81,
                                       8.84, 6.08, 5.39, 8.15, 6.42, 5.73};
    public static final double[] x4 = {8.0, 8.0, 8.0, 8.0, 8.0, 8.0,
                                       8.0, 19.0, 8.0, 8.0, 8.0};
    public static final double[] y4 = {6.58, 5.76, 7.71, 8.84, 8.47,
                                       7.04, 5.25, 12.50, 5.56, 7.91, 6.89};
}
```

然后可以调用任一数组：

```
double[] x1 = Anscombe.x1;
```

A.2 Sentiment

这个**加标签的观点数据集**（sentiment-labeled dataset）来自 https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences。从上述链接下载 3 个文件并把它们放置在 src/main/resources/datasets/sentiment 目录下。它们包含来自 IMDb、Yelp 以及 Amazon 的数据。一个单独句子之后紧跟着用制表符分隔的 0 或 1，分别对应着相应的反对或者同意的观点。但是，并非所有句子都有对应的标签。

IMDb 有 1000 个句子，其中 500 个同意（1），500 个反对（0）。Yelp 有 3729 个句子，其中 500 个同意（1），500 个反对（0）。Amazon 有 15 004 个句子，其中 500 个同意（1），500 个反对（0）。

```
public class Sentiment {

    private final List<String> documents = new ArrayList<>();
    private final List<Integer> sentiments = new ArrayList<>();
    private static final String IMDB_RESOURCE =
    "/datasets/sentiment/imdb_labelled.txt";
    private static final String YELP_RESOURCE =
    "/datasets/sentiment/yelp_labelled.txt";
    private static final String AMZN_RESOURCE =
    "/datasets/sentiment/amazon_cells_labelled.txt";
    public enum DataSource {IMDB, YELP, AMZN};
```

```
public Sentiment() throws IOException {
    parseResource(IMDB_RESOURCE); // 1000个句子
    parseResource(YELP_RESOURCE); // 1000个句子
    parseResource(AMZN_RESOURCE); // 1000个句子
}

public List<Integer> getSentiments(DataSource dataSource) {
    int fromIndex = 0; // 这个索引包括在内
    int toIndex = 3000; // 不包括这个索引
    switch(dataSource) {
        case IMDB:
            fromIndex = 0;
            toIndex = 1000;
            break;
        case YELP:
            fromIndex = 1000;
            toIndex = 2000;
            break;
        case AMZN:
            fromIndex = 2000;
            toIndex = 3000;
            break;
    }
    return sentiments.subList(fromIndex, toIndex);
}

public List<String> getDocuments(DataSource dataSource) {
    int fromIndex = 0; // 这个索引包括在内
    int toIndex = 3000; // 不包括这个索引
    switch(dataSource) {
        case IMDB:
            fromIndex = 0;
            toIndex = 1000;
            break;
        case YELP:
            fromIndex = 1000;
            toIndex = 2000;
            break;
        case AMZN:
            fromIndex = 2000;
            toIndex = 3000;
            break;
    }
    return documents.subList(fromIndex, toIndex);
}

public List<Integer> getSentiments() {
    return sentiments;
}

public List<String> getDocuments() {
    return documents;
}

private void parseResource(String resource) throws IOException {
```

```
        try(InputStream inputStream = getClass().getResourceAsStream(resource)) {
            BufferedReader br =
                new BufferedReader(new InputStreamReader(inputStream));
            String line;
            while ((line = br.readLine()) != null) {
                String[] splitLine = line.split("\t");
                // yelp与amzn[1]的许多句子没有标签
                if (splitLine.length > 1) {
                    documents.add(splitLine[0]);
                    sentiments.add(Integer.parseInt(splitLine[1]));
                }
            }
        }
    }
}
```

A.3　高斯混合

生成服从多维正态分布数据的混合：

```
public class MultiNormalMixtureDataset {
    int dimension;
    List<Pair<Double, MultivariateNormalDistribution>> mixture;
    MixtureMultivariateNormalDistribution mixtureDistribution;

    public MultiNormalMixtureDataset(int dimension) {
        this.dimension = dimension;
        mixture = new ArrayList<>();
    }

    public MixtureMultivariateNormalDistribution getMixtureDistribution() {
        return mixtureDistribution;
    }

    public void createRandomMixtureModel(
    int numComponents, double boxSize, long seed) {
        Random rnd = new Random(seed);
        double limit = boxSize / dimension;
        UniformRealDistribution dist =
            new UniformRealDistribution(-limit, limit);
        UniformRealDistribution disC = new UniformRealDistribution(-1, 1);
        dist.reseedRandomGenerator(seed);
        disC.reseedRandomGenerator(seed);

        for (int i = 0; i < numComponents; i++) {
            double alpha = rnd.nextDouble();
            double[] means = dist.sample(dimension);
            double[][] cov = getRandomCovariance(disC);
            MultivariateNormalDistribution multiNorm =
            new MultivariateNormalDistribution(means, cov);
            addMultinormalDistributionToModel(alpha, multiNorm);
        }
```

注 1：关于 yelp 以及 amzn 的说明，可以参考本书源程序中的 Sentiment.java。——译者注

```
        mixtureDistribution = new MixtureMultivariateNormalDistribution(mixture);
        mixtureDistribution.reseedRandomGenerator(seed);
        // 调用sample()方法将返回相同的结果
    }

    /**
     * 注意这个用于添加内部与外部的已知分布（distros），但是
     * 需要确定将mixture添加mixtureDistribution的简洁方式
     * @param alpha
     * @param dist
     */
    public void addMultinormalDistributionToModel(
    double alpha, MultivariateNormalDistribution dist) {
        // 注意所有的alpha都要进行L1归一化
        mixture.add(new Pair(alpha, dist));
    }

    public double[][] getSimulatedData(int size) {
        return mixtureDistribution.sample(size);
    }

    private double[] getRandomMean(int dimension, double boxSize, long seed) {
        double limit = boxSize / dimension;
        UniformRealDistribution dist =
            new UniformRealDistribution(-limit, limit);
        dist.reseedRandomGenerator(seed);
        return dist.sample(dimension);
    }

    private double[][] getRandomCovariance(AbstractRealDistribution dist) {
        double[][] data = new double[2*dimension][dimension];
        double determinant = 0.0;
        Covariance cov = new Covariance();
        while(Math.abs(determinant) == 0) {
            for (int i = 0; i < data.length; i++) {
                data[i] = dist.sample(dimension);
            }
            // 检查矩阵cov是否奇异……如果是的话，则继续执行
            cov = new Covariance(data);
            determinant = new CholeskyDecomposition(
            cov.getCovarianceMatrix()).getDeterminant();

        }
        return cov.getCovarianceMatrix().getData();
    }

}
```

A.4 Iris

Iris 是著名的包含鸢尾属植物测量值的数据集，它分为 3 类 [2]：

注 2： 相关源代码可以参考 src/main/resources/datasets 路径下的 Iris.java，翻译时进行了相应调整。
——译者注

```
public class Iris {

    private final RealMatrix data;
    private final RealMatrix labels;
    private static final String FILEPATH = "/datasets/iris/iris_data.csv";

    public Iris() throws IOException {

        data = new Array2DRowRealMatrix(150, 4);
        labels = new Array2DRowRealMatrix(150, 3); // 二值化

        try(InputStream inputStream = getClass().
                getResourceAsStream(FILEPATH)) {
            BufferedReader br = new BufferedReader(
                new InputStreamReader(inputStream));
            String line;
            int rowCounter = 0;
            while ((line = br.readLine()) != null) {

                String[] s = line.split(",");
                double sepalLength = Double.parseDouble(s[0].trim());
                double sepalWidth = Double.parseDouble(s[1].trim());
                double petalLength = Double.parseDouble(s[2].trim());
                double petalWidth = Double.parseDouble(s[3].trim());
                String plantClass = s[4].trim();

                data.setEntry(rowCounter, 0, sepalLength);
                data.setEntry(rowCounter, 1, sepalWidth);
                data.setEntry(rowCounter, 2, petalLength);
                data.setEntry(rowCounter, 3, petalWidth);

                if (null != plantClass) switch (plantClass) {
                    case "Iris-setosa":
                        labels.setEntry(rowCounter, 0, 1);
                        break;
                    case "Iris-versicolor":
                        labels.setEntry(rowCounter, 1, 1);
                        break;
                    case "Iris-virginica":
                        labels.setEntry(rowCounter, 3, 1);
                        break;
                    default:
                        System.out.println("something wrong with " +
                            plantClass);
                            break;
                    }

                    rowCounter++;
                }
            }
        }

        public RealMatrix getData() {
            return data;
        }
```

```
    public RealMatrix getLabels() {
        return labels;
    }

}
```

A.5 MNIST

（美国）**国家标准与技术研究所的改良**数据库（MNIST database) 是著名的手写数字数据集，由 70 000 幅数字 0~9 的图片组成。训练集中有 60 000 幅图片，而测试集中有 10 000 幅图片。测试集中前 5000 幅容易识别，而后 5000 幅则难以识别。所有数据都有标签。

文件中所有整数采用**最高有效字节**（MSB，Most Significant Byte）优先（大端）方式存储，这也是绝大多数非 Intel 处理器所使用的整数存储格式。对于 Intel 处理器及其他采用小端方式存储整数的计算机用户而言，必须对整数中的字节顺序进行转换。

该数据库中有 4 个文件。

- train-images-idx3-ubyte：训练集图片
- train-labels-idx1-ubyte：训练集标签
- t10k-images-idx3-ubyte：测试集图片
- t10k-labels-idx1-ubyte：测试集标签

训练集包含 60 000 个实例，测试集包含 10 000 个实例。测试集前 5000 个实例来自原始 MNIST 的训练集，而后 5000 个则来自原始 MNIST 的测试集。测试集前 5000 个实例相对于后 5000 个实例而言更清楚且更容易辨认。

```
public class MNIST {

    public RealMatrix trainingData;
    public RealMatrix trainingLabels;
    public RealMatrix testingData;
    public RealMatrix testingLabels;

    public MNIST() throws IOException {
        trainingData = new BlockRealMatrix(60000, 784); // 图片转为向量
        trainingLabels = new BlockRealMatrix(60000, 10); // 一位有效标签
        testingData = new BlockRealMatrix(10000, 784); // 图片转为向量
        testingLabels = new BlockRealMatrix(10000, 10); // 一位有效标签
        loadData();
    }

    private void loadData() throws IOException {
        ClassLoader classLoader = getClass().getClassLoader();
        loadTrainingData(classLoader.getResource(
        "datasets/mnist/train-images-idx3-ubyte").getFile());
```

```
    loadTrainingLabels(classLoader.getResource(
    "datasets/mnist/train-labels-idx1-ubyte").getFile());
    loadTestingData(classLoader.getResource(
    "datasets/mnist/t10k-images-idx3-ubyte").getFile());
    loadTestingLabels(classLoader.getResource(
    "datasets/mnist/t10k-labels-idx1-ubyte").getFile());
}

private void loadTrainingData(String filename)
throws FileNotFoundException, IOException {
    try (DataInputStream di = new DataInputStream(
    new BufferedInputStream(new FileInputStream(filename)))) {
        int magicNumber = di.readInt(); //2051
        int numImages = di.readInt(); // 60000
        int numRows = di.readInt(); // 28
        int numCols = di.readInt(); // 28
        for (int i = 0; i < numImages; i++) {
            for (int j = 0; j < 784; j++) {
                // 值位于0~255，因此需要归一化
                trainingData.setEntry(i, j, di.readUnsignedByte() / 255.0);
            }
        }
    }
}

private void loadTestingData(String filename)
throws FileNotFoundException, IOException {
    try (DataInputStream di = new DataInputStream(
    new BufferedInputStream(new FileInputStream(filename)))) {
        int magicNumber = di.readInt(); //2051
        int numImages = di.readInt(); // 10000
        int numRows = di.readInt(); // 28
        int numCols = di.readInt(); // 28
        for (int i = 0; i < numImages; i++) {
            for (int j = 0; j < 784; j++) {
                // 值位于0~255，因此需要归一化
                testingData.setEntry(i, j, di.readUnsignedByte() / 255.0);
            }
        }
    }
}

private void loadTrainingLabels(String filename)
throws FileNotFoundException, IOException {
    try (DataInputStream di = new DataInputStream(
    new BufferedInputStream(new FileInputStream(filename)))) {
        int magicNumber = di.readInt(); //2049
        int numImages = di.readInt(); // 60000
        for (int i = 0; i < numImages; i++) {
            // 一位有效编码，0~9列中只有一个是1，其余全部为0
            trainingLabels.setEntry(i, di.readUnsignedByte(), 1.0);
        }
    }
}
```

```
    private void loadTestingLabels(String filename)
    throws FileNotFoundException, IOException {
        try (DataInputStream di = new DataInputStream(
        new BufferedInputStream(new FileInputStream(filename)))) {
            int magicNumber = di.readInt(); //2049
            int numImages = di.readInt(); // 10000
            for (int i = 0; i < numImages; i++) {
                // 一位有效编码，0~9列中只有一个是1，其余全部为0
                testingLabels.setEntry(i, di.readUnsignedByte(), 1.0);
            }
        }
    }

}
```

作者简介

迈克尔·R. 布茹斯托维奇（Michael R. Brzustowicz）曾经是一名物理学家，现在是一名数据科学家，专注于建立分布式数据系统，并从海量数据中提取知识。他大部分的时间都在编写（日常大数据问题的）统计模型方法及机器学习方法的定制多线程代码。他是多家创业公司的合伙人，并在旧金山大学担任兼职教授。

关于封面

本书的封面动物是姬鹬（学名 Lymnocryptes minimus），一种小型的涉禽，分布在大不列颠、非洲、印度，以及地中海周边国家的沿海地区、沼泽、湿草地、泥塘。它们迁徙到北欧以及俄罗斯进行繁殖。

姬鹬是鹬科沙锥属中体型最小的，体长 18~25 厘米，重 34~74 克。它们有斑驳的褐色羽毛以及白色的腹部，在飞行时可以看见纵贯背部的黄色条纹。姬鹬大部分时间生活在水边，在浅水中行走，穿过泥滩去寻找食物，它们的食物包括昆虫、蠕虫、幼虫、植物以及种子。它们的长喙可以帮助它们从地下找到食物。

在求偶期间，雄性姬鹬进行飞行表演，并发出一种类似马蹄哒哒响的声音来呼唤配偶。雌性姬鹬在地上筑巢，产下 3~4 枚鸟蛋。由于它们翅膀的伪装效果佳，以及巢穴的隐蔽性好，因此在野外很难观察到姬鹬。

许多 O'Reilly 图书封面的动物都濒临灭绝，它们都是世界的至宝。想要了解如何提供帮助，可以访问 *animals.oreilly.com*。

封面图片来自 *Wood's Illustrated Natural History*。



微信连接



回复“数据科学”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

Java数据科学实战

数据科学近年来迅速成为了热门研究领域，但很少有数据科学从业人士冒险涉足Java世界。考虑到在工程与科学的交叉地带所需的可伸缩性、稳健性以及便利性，Java其实是一门理想的语言。本书将循序渐进地引导读者进入数据科学的工作流程，在解释数学原理的同时给出代码示例。书中解释了数据科学流程每一步背后的基本数学原理，以及如何采用Java来应用这些原理。

本书内容涉及数据输入与输出、线性代数、统计学、数据操作、学习与预测，以及Hadoop MapReduce在这个过程中所扮演的关键角色。

- **讨论获取数据、清理数据，以及以纯粹方式排列数据的众多方法**
- **理解数据应采用的矩阵结构**
- **学习测试数据来源及数据有效性的基本概念**
- **把数据转换为稳定且可用的数值**
- **理解监督型学习算法与无监督型学习算法，以及评估这些算法是否成功的方法**
- **采用适合数据科学算法的定制组件，设置和运行MapReduce任务**

“这是一本不可多得的、采用Java来实现与应用数据科学的书。书中对机器学习所需的线性代数和统计学的讨论简洁有力。本书汇集了布茹斯托维奇在该领域的丰富实战经验，我强烈推荐。”

——**Terence Parr**
旧金山大学计算机科学
和数据科学教授，
语法分析器生成工具ANTLR之父

迈克尔·R.布茹斯托维奇（Michael R. Brzustowicz），从研究物理的博士后转型为经验丰富的数据科学家，专注于建立分布式数据系统，并从海量数据中提取知识。他大部分的时间都在编写（日常大数据问题的）统计模型方法及机器学习方法的定制多线程代码。他是多家创业公司的合伙人，并在旧金山大学担任兼职教授。

封面设计：**Karen Montgomery** 张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议 计算机/数据科学/Java

人民邮电出版社网址：**www.ptpress.com.cn**

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-53330-2
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks